

**SYSTEM-ON-A-CHIP SOLUTION FOR PLUG AND PLAY
NETWORKED SMART TRANSDUCERS**

by

Gustavo Eduardo Lopez

BS, University of Pittsburgh, 2002

Submitted to the Graduate Faculty of
The School of Engineering in partial fulfillment
of the requirements for the degree of

Master of Science
in
Electrical Engineering

University of Pittsburgh

2004

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This thesis was presented

by

Gustavo Eduardo Lopez

It was defended on

July 16, 2004

and approved by

James T. Cain, Professor, Electrical Engineering

Raymond Hoare, Assistant Professor, Electrical Engineering

Marlin Mickle, Professor, Electrical Engineering

Thesis Advisor: Dr. James T. Cain, Professor, Electrical Engineering

**SYSTEM ON-A-CHIP SOLUTION
FOR PLUG AND PLAY
NETWORKED SMART TRANSDUCERS**

Gustavo Eduardo Lopez, MS

University of Pittsburgh, 2004

The IEEE 1451 standards define sets of common communication interfaces to standardize the connectivity of transducers to microprocessor, instrumentation systems, and networks. This is done by defining different standards that address the various aspects of the development of smart networked transducers. There are seven standards that have been defined so far for the set of standard. The IEEE P1451.0 standard was recently proposed to provide a common set of functions and communication protocols to facilitate interoperability between standards as well as the creation of new standards. The IEEE 1451.1 standard defines a common control network information object model for connecting transducers to Network Capable Application Processors (NCAP). The IEEE 1451.2 defines a Smart Transducer Interface Module (STIM) and a Transducer Electronics Data Sheet (TEDS) for connecting to transducers and NCAPs. The IEEE 1451.3 defines a Transducer Interface Bus Module (TBIM) that is used for transducers that are physically separated but still need to make the connection to the same NCAP. The IEEE P1451.4 will define a mixed-mode transducer interface module. The IEEE P1451.5 will define a protocol for wireless smart sensors. The IEEE P1451.6 recently proposed a standard that will use Consolidated Auto Network (CAN) as the communication medium between the NCAP and the transducers.

Past solutions have concentrated on the implementation of the NCAP and STIM of the 1451.1 and 1451.2 standards. These solutions range from hardware implementations, software implementations, and a combination of hardware and software. However, none of the solutions that have been reported have taken advantage of eliminating inter-chip communications. Our solution eliminates this, and establishes a faster (80 MHz as opposed to the 6000 bits/s that is specified in the 1451.2 standard) and more efficient parallel connection between the NCAP and TIM.

This is implemented using a combination of hardware and software by means of Altera's Excalibur chip. This chip is used because of its ideal structure since it provides a low power embedded processor and a Field Programmable Gate Array (FPGA). Doing this will provide a cost and performance advantage over the separate implementations of the NCAP and TIM, chips or PCBs, assumed by the IEEE 1451 standards.

TABLE OF CONTENTS

1.0	INTRODUCTION	1
1.1	IEEE 1451 OVERVIEW.....	2
1.1.1	IEEE P1451.0.....	4
1.1.2	IEEE 1451.1	4
1.1.3	IEEE 1451.2.....	5
1.1.4	IEEE 1451.3.....	6
1.1.5	IEEE P1451.4.....	7
1.1.6	IEEE P1451.5.....	7
1.1.7	IEEE P1451.6.....	8
2.0	STATEMENT OF PROBLEM.....	9
2.1	BACKGROUND	9
2.2	MOTIVATION AND THE PROBLEM.....	13
2.3	EXCALIBUR SYSTEM.....	14
2.3.1	Peripherals.....	15
3.0	REQUIREMENTS.....	17
3.1	SYSTEM REQUIREMENTS.....	17
3.2	NETWORK CAPABLE APPLICATION PROCESSOR	19
3.2.1	Application Layer	21
3.2.2	Network Layer	22
3.2.3	Transducer Layer	22
3.3	TRANSDUCER INTERFACE MODULE.....	24
3.3.1	Transducer Electronic Data Sheet.....	25
3.3.2	NCAP Communication	25
3.3.3	Trigger.....	26
3.3.4	Status and Interrupts	27
3.4	PHYSICAL WORLD	28

4.0	SPECIFICATIONS	30
4.1	NETWORK CAPABLE APPLICATION PROCESSOR	30
4.1.1	Data Model.....	37
4.1.1.1	Class Header Format and Return Codes	38
4.1.2	Functional Overview and Top-level class definitions	39
4.1.2.1	Block class	40
4.1.2.2	NCAP Block Class.....	43
4.1.2.3	Function Block Class	46
4.1.2.4	Client-Server Network Communication Classes	48
4.1.2.5	Publish-Subscribe Network Communication Classes.....	51
4.1.2.6	Transducer Block Class	53
4.1.2.7	Parameter Classes	57
4.2	TRANSDUCER INTERFACE MODULE.....	59
4.2.1	Transducer Electronic Data Sheet.....	61
4.2.2	NCAP Communication	65
4.2.3	Trigger.....	67
4.2.4	Status and Interrupts	68
4.3	SUMMARY AND TESTS	70
5.0	DESIGN.....	73
5.1	NETWORK CAPABLE APPLICATION PROCESSOR	74
5.1.1	Datatype Mapping.....	76
5.1.2	IEEE 1451.1 API	77
5.1.2.1	Client-Server Network Communications.....	78
5.1.2.2	Publish-Subscribe Network Communications	83
5.1.3	Transducer I/O API.....	86
5.1.3.1	Transducer Block Class	86
5.1.3.2	Parameter Classes	90
5.1.4	Application Layer	94
5.1.4.1	Block Class	94
5.1.4.2	NCAP Block Class.....	98
5.1.4.3	Function Block Class	102

5.1.5	Summary and Example implementation	105
5.2	TRANSDUCER INTERFACE MODULE	111
5.2.1	Transducer Electronic Data Sheet	112
5.2.2	Transducer Interface Module Control Unit	113
5.2.2.1	Example Configuration	120
5.2.3	Transducer Channel Block	129
5.2.4	Interrupt Management	147
5.2.4.1	Interrupt Service Routine	150
5.2.5	Summary and Constraints	151
6.0	IMPLEMENTATION AND TESTS	153
6.1	APPLICATION SYSTEM	153
6.2	SYSTEM IMPLEMENTATION	159
6.2.1	Network Capable Application Processor Implementation	162
6.2.2	Transducer Interface Module Implementation	165
6.3	TEST RESULTS	174
7.0	CONCLUSIONS AND FUTURE WORK	182
7.1	CONCLUSIONS	182
7.2	FUTURE WORK	183
	APPENDIX A. NCAP OBJECT MODEL	185
	APPENDIX B. NCAP SOFTWARE CODE	192
	APPENDIX C. TIM VHDL CODE	196
	APPENDIX D. GLUE LOGIC VHDL CODE	221
	APPENDIX E. TEDS BLOCKS	239
	BIBLIOGRAPHY	242

LIST OF TABLES

Table 3-1 TIM's Status Bits.....	27
Table 4-1 Simple Primitive Types	37
Table 4-2 Class Header Format	38
Table 4-3 Meta-TEDS Structure	61
Table 4-4 Channel-TEDS Structure.....	63
Table 4-5 TIM's Commands.....	66
Table 4-6 System Summary	70
Table 5-1 Datatype to C Mapping	77
Table 5-2 AHB Signal Summary	115
Table 5-3 AMBA AHB Command Set.....	119
Table 5-4 Status Register Bits.....	133
Table 6-1 Thermistor's ADC Output and Corresponding Temperature Value	157
Table 6-2 Interrupt Service Routines summary	174
Table 6-3 Synthesis Results	174

LIST OF FIGURES

Figure 1-1 IEEE 1451 Structure Overview.....	2
Figure 1-2 STIM Definition ⁽³⁾	6
Figure 1-3 TBIM Overall Structure ⁽⁴⁾	7
Figure 2-1 Excalibur System Architecture	14
Figure 3-1 Design Reference Structure.....	18
Figure 3-2 IEEE 1451.1 NCAP	20
Figure 3-3 TIM trigger behavior.....	27
Figure 4-1 UML Model of IEEE 1451.1 Objects	31
Figure 4-2 Conceptual View of an IEEE 1451.1 NCAP	40
Figure 4-3 Block Class UML Diagram.....	40
Figure 4-4 State Machine for Block Class.....	43
Figure 4-5 NCAP Block UML Class Diagram	43
Figure 4-6 State Machine for the NCAP Block	46
Figure 4-7 Function Block UML Class Diagram	47
Figure 4-8 State Machine for a Function Block.....	48
Figure 4-9 Entity UML Class Diagram	48
Figure 4-10 Client Port UML Class Diagram.....	49
Figure 4-11 Client-Server Communication Model	51
Figure 4-12 Publisher Port UML Class Diagram	51
Figure 4-13 Subscriber Port UML Class Diagram	52
Figure 4-14 Transducer Block UML Class Diagram.....	54
Figure 4-15 Correction state machine for a Transducer Block.....	55
Figure 4-16 Model of a TIM and Transducer Block.....	56
Figure 4-17 Parameter with Update Class UML Diagram	57
Figure 4-18 Time sequence behavior of UpdateAndRead and WriteAndUpdate	58
Figure 4-19 Physical Parameter UML Class diagram	58

Figure 4-20 TIM Overall Structure.....	60
Figure 4-21 Interrupt Masking.....	69
Figure 5-1 NCAP Overview	74
Figure 5-2 Class Hierarchy Implementation.....	75
Figure 5-3 Structure instantiation in C.....	76
Figure 5-4 Entity Class Structure in C.....	78
Figure 5-5 Perform() Operation pseudo-code.....	81
Figure 5-6 Client Port Class Structure in C	81
Figure 5-7 Execute() Pseudo-Code	82
Figure 5-8 Publisher Port Class Structure in C.....	83
Figure 5-9 Subscriber Port Class Structure in C.....	84
Figure 5-10 Transducer Block Class Structure in C	86
Figure 5-11 EnableCorrections() operation pseudo-code	87
Figure 5-12 DisableCorrections() operation pseudo-code.....	88
Figure 5-13 UpdateAll() Pseudo-code	90
Figure 5-14 Parameter With Update Class Structure in C.....	91
Figure 5-15 UpdateAndRead() pseudo-code	91
Figure 5-16 WriteAndUpdate() Pseudo-code	92
Figure 5-17 Physical Parameter Class Structure in C.....	93
Figure 5-18 Block Class Structure in C	95
Figure 5-19 GoActive() operation pseudo-code	97
Figure 5-20 GoInactive() operation pseudo-code	97
Figure 5-21 Initialize() operation pseudo-code.....	98
Figure 5-22 NCAP Block Structure in C	99
Figure 5-23 Function Block Class Structure in C.....	102
Figure 5-24 Start() Operation pseudo-code	103
Figure 5-25 Clear() Operation pseudo-code	103
Figure 5-26 Pause() Operation pseudo-code.....	104
Figure 5-27 Resume() Operation pseudo-code	104
Figure 5-28 Application-specific example operation SampleAndSetAll().....	106
Figure 5-29 TIM Commands Header File	108

Figure 5-30 Snippet C Code for UpdateAndRead Operation	109
Figure 5-31 Snippet C Code for WriteAndUpdate Operation	110
Figure 5-32 TIM high-level Architecture	111
Figure 5-33 Simple AMBA transfer ⁽¹⁹⁾	114
Figure 5-34 PLD Slave and AHB Connection.....	114
Figure 5-35 State Machine for the Control Unit.....	116
Figure 5-36 Pseudo VHDL code for TIM Control Unit	117
Figure 5-37 Control Unit Example Configuration for Write Interrupt Mask Command	120
Figure 5-38 Snippet VHDL code for TIM Control unit for Write Interrupt Mask Command ...	121
Figure 5-39 Control Unit Example Configuration for Read Interrupt Mask Command	122
Figure 5-40 Snippet VHDL code for TIM Control unit for Read Interrupt Mask Command	122
Figure 5-41 Control Unit Example Configuration for Status Command.....	123
Figure 5-42 Snippet VHDL code for TIM Control unit for Status Command	124
Figure 5-43 Control Unit Example Configuration for Trigger Command	125
Figure 5-44 Snippet VHDL code for TIM Control unit for Trigger Command	125
Figure 5-45 Control Unit Example Configuration for Write Actuator Data Command.....	126
Figure 5-46 Snippet VHDL code for TIM Control unit for Write Actuator Data Command	127
Figure 5-47 Control Unit Example Configuration for Read Sensor Data Command.....	127
Figure 5-48 Snippet VHDL code for TIM Control unit for Read Sensor Data Command	128
Figure 5-49 Control Unit Example Configuration for Reset Command.....	129
Figure 5-50 Snippet VHDL code for TIM Control unit for Reset Command	129
Figure 5-51 TIM Transducer Channel	131
Figure 5-52 Sensor Channel Interface with an 8-bit ADC Configuration	132
Figure 5-53 “Glue” Logic for Sensor Channel interfaced with ADC.....	133
Figure 5-54 VHDL Code for Status Generation Block for a sensor channel that is interfaced with an 8-bit ADC.....	134
Figure 5-55 “Glue” logic State Machine for Sensor Channel interfaced with ADC	135
Figure 5-56 Sensor Channel Interface with digital I/O Configuration	136
Figure 5-57 “Glue” Logic for Sensor Channel interfaced with digital I/O.....	137
Figure 5-58 “Glue” logic State Machine for Sensor Channel interfaced with digital I/O.....	138
Figure 5-59 Actuator Channel Interface with an 8-bit DAC Configuration.....	139

Figure 5-60 “Glue” Logic Architecture for an Actuator Channel interfaced with DAC.....	140
Figure 5-61 “Glue” logic State Machine for Actuator Channel interfaced with DAC.....	141
Figure 5-62 Actuator Channel Interface with digital I/O Configuration	142
Figure 5-63 TIM Transducer CHANNEL_ZERO Structure	143
Figure 5-64 CHANNEL_ZERO Example for 8-Channel TIM Configuration.....	144
Figure 5-65 “Glue” Logic for CHANNEL_ZERO.....	145
Figure 5-66 State Machine for “Glue” logic’s control.....	147
Figure 5-67 Interrupt Controller overall structure	148
Figure 6-1 Thermistor’s Connections	154
Figure 6-2 Thermistor’s behavior plots	155
Figure 6-3 ADC Voltage dividers.....	156
Figure 6-4 Photocell’s connections.....	158
Figure 6-5 LED Connections	158
Figure 6-6 Picture of the Implementation.....	159
Figure 6-7 System Implementation.....	160
Figure 6-8 System’s Embedded Stripe	164
Figure 6-9 Execute() C Code	165
Figure 6-10 Top Level System Implementation	166
Figure 6-11 NCAP/TIM Connections, part (a) from Figure 6-10	167
Figure 6-12 Priority encoder connections with Transducer channels, part (b) of Figure 6-10...	168
Figure 6-13 Transducer Channel Connections with Control Logic, part(c) of Figure 6-10.....	169
Figure 6-14 Transducer channels and glue logic, part (d) of Figure 6-10	171
Figure 6-15 Interface between Control Unit, Transducer Channels and “glue” logic, part (e) of Figure 6-10.....	172
Figure 6-16 Initialization Results.....	176
Figure 6-17 Application Code for Individual Triggers and Client-Server Operations.....	177
Figure 6-18 Main Loop first iteration	179
Figure 6-19 Application Code for Global Trigger and Publish-Subscribe Operations	180
Figure 6-20 Main Loop Second Iteration.....	180

1.0 INTRODUCTION

Transducers, defined as devices that convert energy from one domain into another (either sensors or actuators), are frequently used in the manufacturing, industrial control, automotive, aerospace, building, and biomedical industries among others. Because of the diversity of the transducer market, manufacturers are always looking for ways to build low-cost, networked smart transducers. The Institute of Electrical and Electronics Engineers (IEEE) in conjunction with the National Institute of Standards and Technology (NIST) addressed this issue by creating a family of standards to aid in the design and development of smart networked transducers.

The ultimate goal of the standards is to achieve transducers to network interchangeability and transducer to networks interoperability. This is done by defining a set of common communication interfaces for connecting transducers to microprocessors, instruments and field networks.

This thesis concentrates on the design and implementation of a single chip solution of the approved IEEE 1451 standards, in which inter-chip communications are eliminated yielding a speed improvement over the implemented solutions to date.

1.1 IEEE 1451 OVERVIEW

The standards were first proposed in September 1993, when NIST and the IEEE's Technical Committee on Sensor Technology of the Instrumentation and Measurement Society co-sponsored a meeting to discuss smart sensor communication interfaces and the possibility of creating a standard interface. The response was to establish a common communication interface for smart transducers. Since then, a series of workshops has been held and seven technical working groups have been formed to address different aspects of the interface standard.

This family of standards is designed to work in concert with each other to ease the connectivity of sensors and actuators into a device or field network. The overall structure for the standards is shown in Figure 1-1.

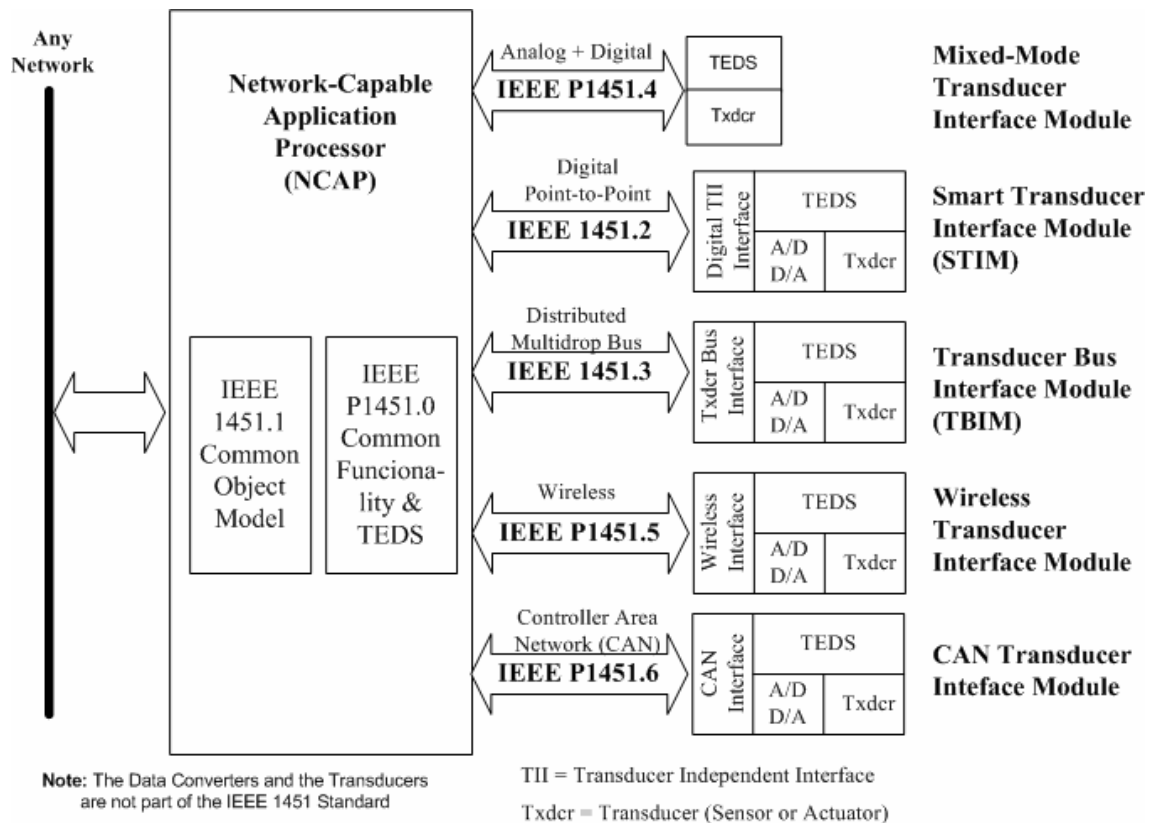


Figure 1-1 IEEE 1451 Structure Overview

From the figure, it is easy to see that transducers are interfaced to a Transducer Interface Module (TIM) that is denoted smart because it provides functions like self-identification. The TIM is controlled by a Network Capable Application Processor (NCAP) in a network-independent environment. Depending on the standard, single or multiple TIMs can be connected to a single NCAP.

Each TIM defines a Transducer Electronic Data Sheet (TEDS) and a communication interface (analog, digital or wireless) for connection with an NCAP. The TEDS allows for self-identifying transducers, thus enabling “plug and play”. The communication interface allows the NCAP to access and control the TIM.

The most important achievement of the standards is the idea of “plug and play” smart networked transducers which happens at both the TIM and NCAP level. As it has been mentioned before the TEDS enables “plug and play” for the TIM. The object model defined for the NCAP also gives this advantage by defining an Application Programming Interface (API) for NCAP to Transducer and NCAP to Network communications. These APIs allow for a common network-independent application model that maps to any transducer network protocol establishing interoperability between transducers and existing control network.

The following sub-sections of this Chapter go into detail about the different standards and their current status.

1.1.1 IEEE P1451.0

This standard ^{(1)*} will provide a common set of functions, communications protocols, and TEDS formats that facilitate interoperability among the family of standards. It will also simplify the creation of future standards for different physical interfaces while maintaining interoperability among the family members. This standard is in the early stages of development.

1.1.2 IEEE 1451.1

This standard defines an interface for connecting NCAPs to control networks through the development of a common control network information object model for smart sensors and actuators ⁽²⁾. The purpose of the standard is to provide a network-neutral application model that will simplify the interface of smart sensors and actuators to a network. This way, at the application level, the physical connections become transparent to the user.

The NCAP is defined by hardware and software blocks. The software blocks are defined by APIs that hide the communication details to a particular network or transducer communication interface. The hardware blocks are composed of the IO and network hardware needed for the various TIMs and networks respectively.

The NCAP is divided into three layers (Network, Application, and Transducer). The Network and Transducer layers handle the communication and interface to both the network and the various TIMs implemented in the system.

An information model describes the objects in the standard. This model is defined by a software architecture that includes an Object model (for the software components of IEEE1451.1

* Parenthetical references placed superior to the line of text refer to the bibliography.

systems), a data model (for the information communicated across the specified object interfaces), and two network communication models (Client-Server, and Publish/Subscribe methods).

1.1.3 IEEE 1451.2

This was the first published standard ⁽³⁾ in the family. However, it has not been widely accepted in industry because of discontent with the digital communication interface and the complex software features for its interface with an NCAP.

This standard defines a STIM, TEDS, and Transducer Independent Interface (TII) for NCAP communication. Each Transducer is denoted in the STIM as a channel, and there can be up to 255 channels within one STIM. Individual channels and the STIM as a whole can be accessed by the NCAP. The different transducers that are interfaced to the STIM are triggered (sampled or set) by a command that is sent from the NCAP to the STIM. The STIM decodes this information and then sends the results back to the NCAP.

The TII is a ten wire serial connection for NCAP/STIM communication and has been widely criticized by industry because of its complexity, so as part of this standard's revision; the TII may be eliminated in favor of a RS-232 serial connection. Figure 1-2 shows the structure of the STIM.

The TEDS supports a variety of transducers and is accessed by the digital interface (TII). The TEDS can be written by the NCAP or it can also be set at manufacture time. It resides in non-volatile memory, and contains fields that describe the type, attributes, operation, and calibration of the transducers. The TEDS is the core of the standard since it provides a method for self-identifying transducers.

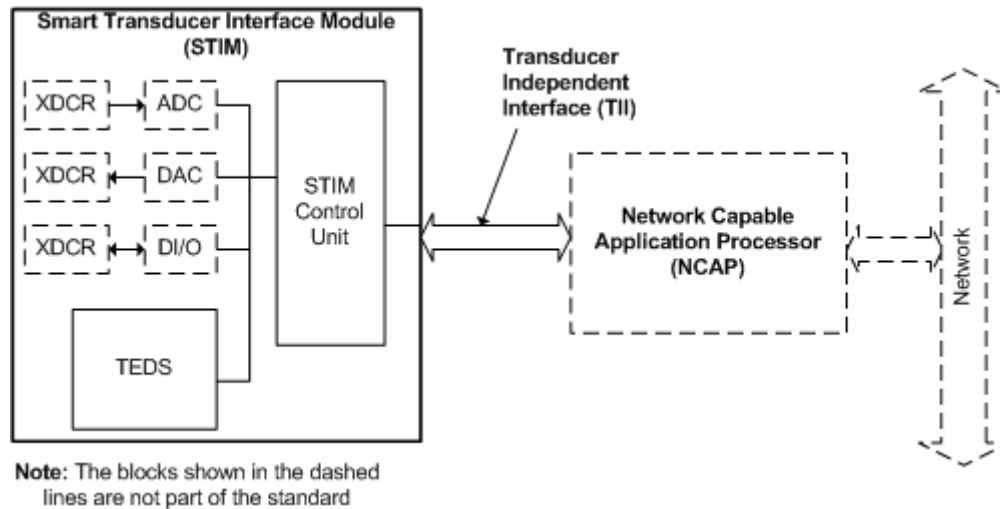


Figure 1-2 STIM Definition ⁽³⁾

1.1.4 IEEE 1451.3

This standard ⁽⁴⁾ introduces the concept of a Transducer Bus Interface Module (TBIM) and a Transducer Bus Controller (TBC) connected by a Transducer Bus. A TBIM contains the bus interface, TEDS, and the interface to the transducers. The TBC is the hardware and software in the NCAP or host processor that provides the interface to the Transducer Bus. The Transducer Bus provides a communications path between an NCAP or host processor and one or more TBIMs.

This standard was created for transducers that are physically separated but still need to connect to a single NCAP (something that is not supported in the IEEE 1451.2). The different communication between the various TBIMs and single NCAP is synchronized by a sync signal that is handled by the TBC. Figure 1-3 shows the structure of the standard.

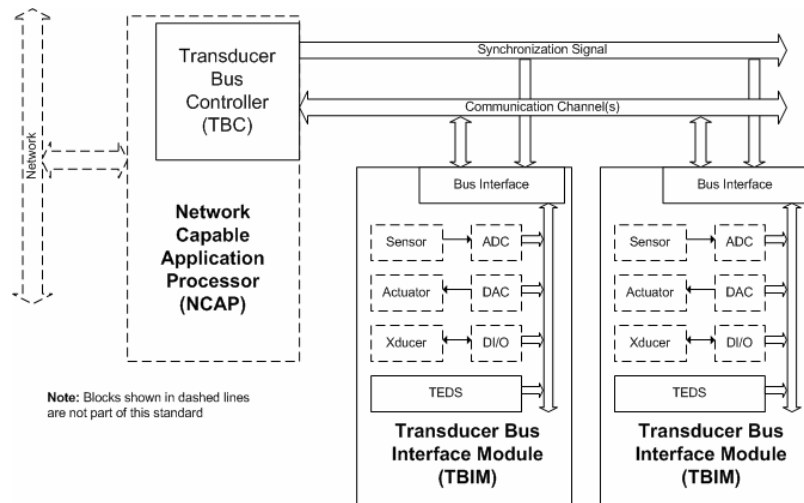


Figure 1-3 TBIM Overall Structure ⁽⁴⁾

1.1.5 IEEE P1451.4

This standard ⁽⁵⁾ will define a mixed-mode transducer interface able to work both in analog signal transmission mode and in digital communication mode, but not simultaneously. An IEEE P1451.4 transducer contains a Mixed-Mode Interface (MMI) and a TEDS. The MMI is a two-wire, master-slave, multi-drop, serial connection. The TEDS resides in one or more memory nodes on the MMI. Its main objective is to make a bridge between legacy transducers and the networked smart transducers. This standard is in the balloting stage at the moment and could be approved by the end of 2004.

1.1.6 IEEE P1451.5

This standard ⁽⁶⁾ will define an interface for wireless communications and data formats for transducers, and a TEDS based on the IEEE 1451 concept, and protocols to access TEDS and transducer data.

The proposed standard will include multiple MAC/PHY combinations so it will be easy to implement any type of wireless network as a NCAP-TIM interface. Different physical layers are being analyzed for the implementation of the standard. Among the physical layers being considered are the IEEE 802.11 (WiFi), the IEEE 802.15.1 (Bluetooth), the IEEE 802.15.4 (LR-PAN, lower power, lower rate, lower cost), and other proprietary layers. This standard is in the stages of development.

1.1.7 IEEE P1451.6

This project ⁽⁷⁾ establishes a CANopen-based network for multi-channel transducer modules. The standard defines the mapping of IEEE 1451 Transducer Electronic Data Sheet (TEDS) to the CANopen dictionary entries as well as communication messages, process data, configuration parameters, and diagnosis information. It adopts the CANopen device profile for measuring devices and closed-loop controllers. This project defines an intrinsically safe (IS) CAN physical layer.

The targets of the standard are sensor bus and transducer network users across various industries. In particular, the instrumentation and measurement, and process control industry. CANopen network will be able to use IEEE 1451 transducers and have the benefit of the TEDS. Only a proposal has been submitted for this standard.

2.0 STATEMENT OF PROBLEM

The various solutions that have been designed and implemented based on the 1451 standards address different areas in the development and understanding of this family. However, none of the implementations has taken advantage of a single chip solution by eliminating inter-chip communications. This inter-chip communication can be eliminated because there is no physical separation between the NCAP and TIM, and a high-speed connection between the modules can be established. This issue is addressed in this thesis by designing the NCAP/TIM combination using Altera's Excalibur chip.

2.1 BACKGROUND

Most of the implementations that have been documented on the standards have been on the implementation of a STIM (1451.2) and NCAP (1451.1) since they were the first standards to be approved. In this section we will review reported implementations on the IEEE 1451 family.

Analog Devices ⁽⁸⁾ designed a microcontroller (ADuC812 ⁽⁹⁾) chip specifically to meet the IEEE 1451.2 standard requirements. This chip integrates the data converters (for transducer interfacing), EEPROM (for the TEDS), and an 8-bit microcontroller for the STIM's functionality ⁽¹⁰⁾.

The first implementation that we will review is from Dr. Paul Conway and his research group from the University of Limerick, Ireland. This research group designed and implemented an IEEE 1451.2 STIM using Analog Device's ADuC812 microconverter ⁽¹¹⁾. The motivation for the solution was to exploit the on-chip resources provided by the ADuC812. In order to do this, software was developed to map the STIM's functionality to the microconverter. On-chip data converters and flash/EE memory were used for transducer interfacing and TEDS (only mandatory blocks were implemented) respectively. The software-architecture was designed to meet all the mandatory specifications of the standard. The entire implementation for a two-channel STIM consumed 5,534 bytes of program memory, 178 bytes of RAM memory and 268 bytes of flash/EE data memory. This solution found that a STIM implementation using the ADuC812 is useful only when implementing a small system, because of limitations of on-chip resources, such as the 640 bytes of flash memory which is too small to hold an entire set of TEDS, if mandatory (Meta and Channel TEDS) and optional (Calibration information) TEDS blocks are to be included in a system with more than one transducer.

Another solution that has been implemented is a proof-of concept IEEE 1451.1 design in VHDL connecting to a Bluetooth wireless network ⁽¹²⁾. This research group, from Ohio State University, developed a network infrastructure (defined by a software model) that enables smart transducer communication through Bluetooth using the OBEX Session protocol. This protocol is specific to Bluetooth and is used for wireless object exchange. This network infrastructure acted as a bridge between the 1451.1 data types and operations to the network-specific OBEX format on both the client and server sides. This was implemented and simulated using behavioral VHDL. This experiment relied completely on simulations. Future work for this project will include a synthesizable version of the hardware representation that can be mapped to an

Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA). This type of solution is helpful for future versions of smart transducers that have the capability of hosting a Java Virtual Machine (JVM), in which the VHDL design may be translated to a high-level description.

Next, we discuss a full hardware implementation of a 1451.2 STIM ⁽¹³⁾, which was developed in the Nuclear Physics Institute of Lyon in France. In this project, a STIM was designed in VHDL and synthesized onto an Altera APEX20K FPGA. The transducers and the NCAP were interfaced directly to the FPGA. The NCAP/STIM communication was done via the TII. This solution allowed the concept of Ethernet Capable Front-end Module (frequently used for slow control-type sensors) to include the data-acquisition requirements of a high-energy physics experiment. This solution achieved data rates close to 1Mbps on the Ethernet port.

An interesting solution involved the development of a STIM and NCAP (connecting through the TII) for a CAN network ⁽¹⁴⁾. Here, the University of Barcelona research group designed a two channel STIM based on a microcontroller that includes a multiplexed eight channel A/D converter, and an I²C bus interface. The TEDS was implemented using a serial EEPROM that interfaced directly to the microprocessor. They also developed a software tool for the generation of a TEDS storing it in the serial EEPROM. The 1451.1 information model was also designed and adapted to work in a CAN network environment. This solution was the first one to modify/generate the TEDS *in situ* via a software tool that was built.

The implementation of STIMs in industry has been slow due to discontent with the communication protocol (TII), which is believed to be overly complicated. Dr. Darold Wobschall from the State University of New York at Buffalo addressed this issue by exploring different connections between a STIM and an NCAP ⁽¹⁵⁾. The different communication protocols

that were implemented were RS232, RS485, TII, Microlan/1-wire, IEEE 1451.4, Eibus, and I²C. Users can select any one of the protocols for the NCAP/STIM communication. There were no comparisons on how each communication protocol performed. This was because the objective of the project was to design a device that would give users a broader range of selection in the communication interface to smart transducers.

One of the most interesting solutions implemented presented a System on a Chip solution for the STIM. This solution was developed by Dr. Angel de Castro and his research group at the University of Madrid, Spain. The STIM and NCAP were completely described in hardware (Xilinx Virtex XCV800 FPGA) using VHDL ^(16,17). This design is unique in that a microcontroller was not used for the STIM's functionality. This provides a speedup advantage (hardware ran at 45 MHz) over the usual software solutions. Future work in this group consists of an ASIC solution, in which the data converters and Micro Electro-Machine Systems (MEMS) sensors can be integrated in the SOC design. To date, there has been no documentation on the success or completion of that work.

This last solution is close to what we are trying to accomplish in this thesis, since they designed the NCAP and TIM in a single chip and targeted it for an ASIC solution. However, they did not eliminate inter-chip communications and the information model was not built for the 1451.1 NCAP, both of which are key elements in the design that is presented here.

2.2 MOTIVATION AND THE PROBLEM

A single chip solution would provide a cost and performance advantage over the separate implementations of the NCAP and Transducer Interface Module (TIM), chips or PCBs, assumed by the IEEE 1451 standards. Advances in integrated circuit technology have allowed systems with the complexity of both devices to possibly be implemented on a single chip. In this way inter-chip communications can be eliminated, increasing the speed of the TIM/NCAP connection paving the way for faster and more efficient instrumentation and control systems. This was the motivation for the effort to develop a 1451 compatible single chip implementation of a combined NCAP and TIM.

The goal of this thesis is to design and implement an IEEE 1451 single chip solution using Altera's Excalibur chip. The TIM will have the functionality of a smart transducer as is stated in the standards. The information model for the NCAP will be designed using the functionality described in the IEEE 1451.1 standard.

The most important aspect of the problem will be the elimination of inter-chip communication since it is a solution that has not been explored. In achieving this solution, we will provide academia and industry with a more efficient (performance-wise) connection between the NCAP and TIM. This implementation will also give another option and better understanding on the family of standards. Next, we will give an overview of the Excalibur system and what it provides.

2.3 EXCALIBUR SYSTEM

The Excalibur chip ⁽¹⁸⁾ is composed of a Reduced Instruction Set Computer (RISC) processor (ARM 922T) with programmable logic on a single device. The processor communicates to its peripherals by means of an Advanced Microcontroller Bus Architecture (AMBA) high-performance bus (AHB). The chip's architecture provides a variety of on-chip peripherals such as an interrupt controller, programmable logic device (PLD), etc. Figure 2-1 shows the chip's architecture.

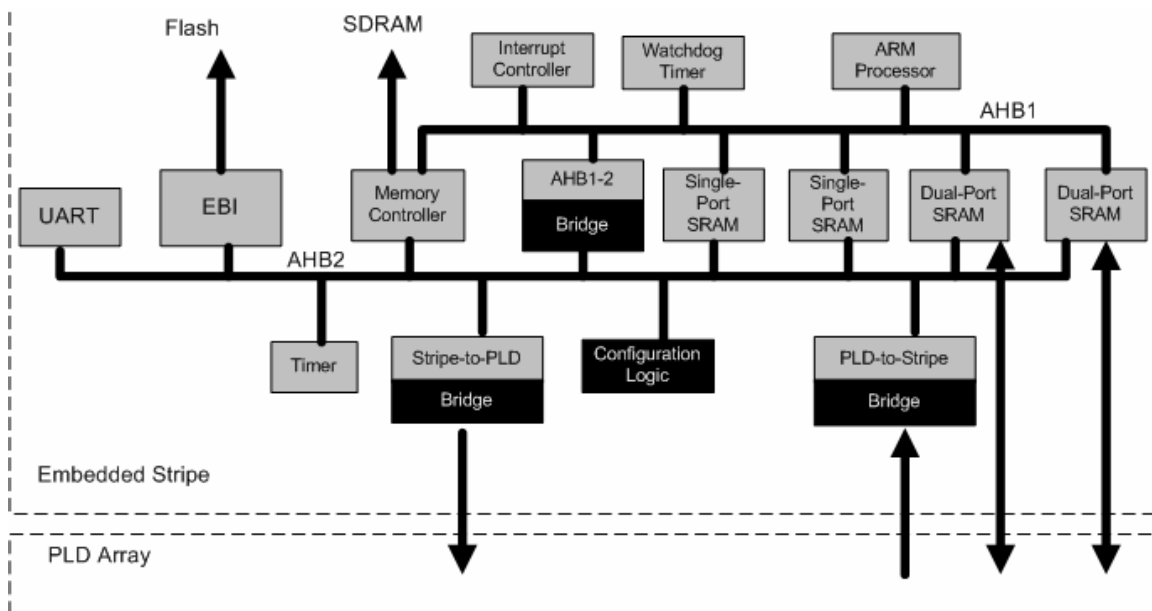


Figure 2-1 Excalibur System Architecture

The bus architecture used by the Excalibur family conforms to specifications of the AMBA bus ⁽¹⁹⁾. Two AMBA-compliant AHBs ensure that the embedded processor activity is unaffected by peripheral and memory operation. Three bidirectional AHB-to-AHB bridges enable embedded peripherals and PLD-implemented peripherals to exchange data with the embedded processor or with other peripherals. The bidirectional bridges handle the

resynchronization across the domains and are capable of supporting 32-bit data accesses to the entire 4-Gbyte address range (32-bit address bus).

The only bus master on AHB1 is the ARM processor. Processor-specific slaves such as the interrupt controller are local to the AHB1. Memory resources such as the on-chip SRAM are also local to the AHB1, allowing for fast access to the memory by the embedded processor.

Any transaction that after decoding is not intended for AHB1 is then routed to the AHB1-2 bridge. This bridge is a slave on AHB1, giving the embedded processor access to AHB2. There are three bus masters on AHB2 (ARM processor, Configuration logic, and PLD). Note that the PLD can be configured as either a master or a slave on AHB2. A priority arbitration scheme is used to grant access to masters on the AHB2 bus.

The ARM embedded processor supports both the 32-bit ARM and 16-bit Thumb instruction sets. It consists of a Harvard architecture, implemented using a five-stage pipeline. It allows for single clock-cycle instruction operation through simultaneous fetch, decode, execute, memory, and write stages ⁽²⁰⁾.

2.3.1 Peripherals

The embedded stripe contains a variety of peripherals that can be configured in different ways depending on the application. These peripherals include: Configuration Registers, Embedded Stripe Phase-Locked Loops (PLLs), Universal Asynchronous Receiver Transmitter (UART), Timer, Watchdog timer, General Purpose I/O Port, Interrupt Controller, PLD, and an External Bus Interface (EBI).

The interrupt controller provides a simple, but flexible, software interface to the interrupt system. It can be configured to handle up to 64 individual interrupts. Designers can also build their own interrupt controller in case more interrupts need to be handled.

The programmable logic device that is provided comes in various sizes depending on different versions of the chip. The device can be configured to implement any custom hardware logic.

The EBI is a 16-bit bidirectional memory interface that provides a bridge between external devices (flash memory, or memory mapped devices) and the AHB2 bus. The EBI supports up to four blocks of up to 32 Mbytes of external memory or memory mapped devices of different configurations.

The rest of the thesis is structured as follows. Chapter 3 defines the functional and non-functional requirements of the system in order to remain IEEE 1451 compliant. Chapter 4 entails the specifications derived from the requirements section. Chapter 5 consists of the design decisions made from the different discussions of requirements and specifications. Chapter 6 includes the implementation along with the results of the tests. Lastly, Chapter 7 discusses the conclusions and future work that can be done regarding the problem.

3.0 REQUIREMENTS

In order to design an IEEE 1451 compliant system, a variety of requirements must be met. These requirements can be divided into three sub-sections, system Requirements, for the application-specific behavior and object interaction, NCAP requirements, for the top-level functionality and structure of a 1451.1 NCAP, and TIM requirements, for the top-level functionality and structure of an IEEE 1451 compliant TIM.

The common design requirement among the system's objects is that they shall be designed in a network and transducer independent environment. The design shall focus on the IEEE 1451.1 standard for the NCAP's functionality, and the 1451.2 and 1451.3 standards for the functionality of the TIM. Note that during this chapter we will use the word "shall" to denote a requirement. This is done to use the same wording as in the IEEE 1451 standards.

3.1 SYSTEM REQUIREMENTS

The system requirements are both functional and non-functional. The standards do not specify non-functional requirements because they are implementation and application specific, as it only specifies the functionality and not how it is to be implemented. Since one of the main goals of this project is to eliminate inter-chip communications, the results should obtain higher speeds for

the NCAP/TIM communication. This implementation shall have the non-functional requirement of achieving higher data rates than the 6000 bits/s maximum rate of the TII defined for the 1451.2 STIM⁽³⁾.

The functional requirements encompass the system's behavior as stated by the IEEE 1451 family. The main requirement is that the design shall be network and transducer independent. This means, that at the application level, the physical connections in the system become transparent to the users.

This network and transducer independency is achieved by different blocks that are defined in the standard. These blocks shall be designed and implemented as is shown in Figure 3-1.

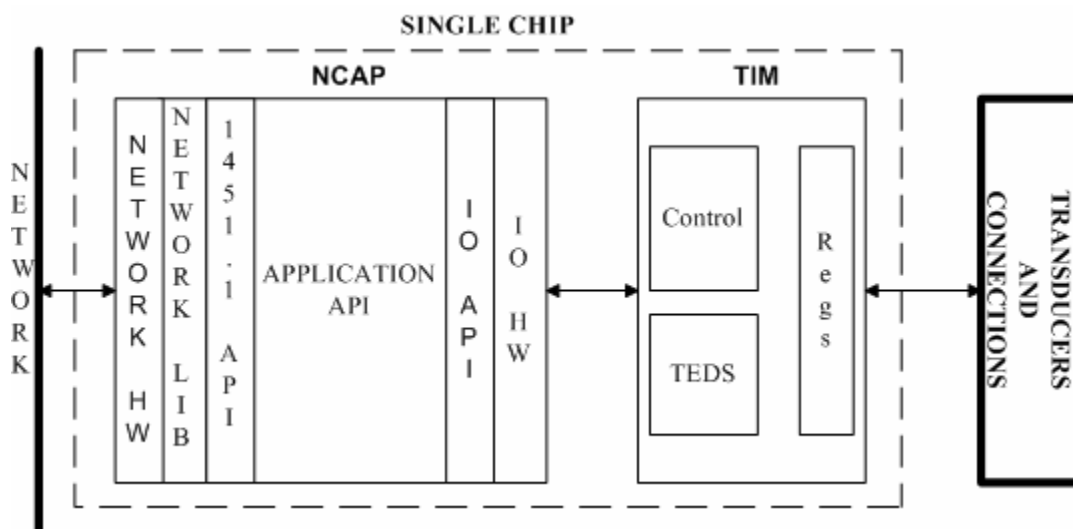


Figure 3-1 Design Reference Structure

The Network Hardware, Network Library and IO Hardware objects (shown in Figure 3-1) are defined in the standard as implementation-specific, so their functionality is not specified by the standard as it depends on the application. However, these blocks shall interface with the APIs that are shown in the figure and meet the electrical specifications of the underlying network

and the Excalibur chip. The remaining blocks (1451.1 API, Application API, IO API, Control, TEDS, and Registers) that are shown within the single chip annotation in Figure 3-1 are completely described in the standard and shall be designed following its specifications. The transducers and the network are application specific and they are not defined by the standard and they cannot be implemented within the FPGA. However, they too shall meet the electrical specifications of the Excalibur chip.

The design shall prove its compliance with the standard by showing interactions between a network, NCAP and TIM. These interactions shall make use of the objects and formats described in the standards.

3.2 NETWORK CAPABLE APPLICATION PROCESSOR

The Network Capable Application Processor (NCAP) consists of hardware and software blocks that shall be implemented in order to remain compliant with the IEEE 1451.1 standard. An information model is defined for the NCAP. This information model consists of a data model (for the datatypes), an object model (for the classes), and two communication models (used for network communications). The complete information model shall be implemented. In order to do this, we shall use Figure 3-2 as a reference for the NCAP design.

Note that the NCAP is divided into three layers, network, application, and transducer. For the network layer, the hardware shall consist of the network-specific logic that is needed for the underlying network. Also, the software for this layer shall consist of the network library (network-specific), and the 1451.1 API (standard-defined).

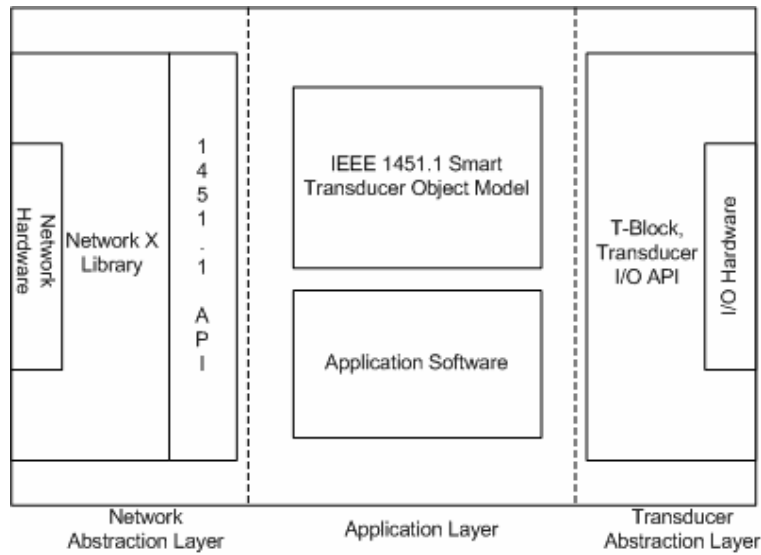


Figure 3-2 IEEE 1451.1 NCAP

The application layer shall only consist of software blocks. These blocks shall consist of the standard-defined blocks for the application, as well as application-specific code. For the transducer layer, the hardware shall consist of the specific hardware needed to communicate with the TIM. The software shall consist of the T-Block API.

The hardware blocks that have been defined for the NCAP shall be designed to meet the electrical and timing specifications of the underlying network as well as those of the Excalibur chip. On the other hand, the software blocks shall be designed using the objects and formats defined in the 1451.1 standard. It is important to note, that for the software some blocks are standard-defined while others are implementation-specific. However, these objects shall communicate with each other using the object orientation of the 1451.1 standard. The following discussion gives background information on the objects of an IEEE 1451.1 NCAP.

The objects defined by the information model are at the core of the standard, because they allow for a common interface that is network and transducer independent. The common control network information object model defines three major blocks for the standard's API.

These blocks include an NCAP Block that consists of a standard software interface for supporting network communications and system configuration (key source for network communications and system configuration), Transducer Blocks for the interface between transducers and application functions, and Function Blocks which encapsulate application-specific functionality.

The three major blocks shall work together to communicate with the “physical world” consisting of the networks and transducers. For network communications, these objects can be either network visible or independent. Network visible means that the object can be directly accessed by a network, while independent means that the blocks may only be accessed by other objects within that local NCAP. For an object to become network visible it shall be registered with its local NCAP block. If they are not registered then they shall not be accessed directly by the network but rather by the NCAP application.

3.2.1 Application Layer

The application-specific behavior of the NCAP shall be defined within this layer. The application’s behavior shall not be affected by the different physical connections that can be made between the NCAP/network and NCAP/TIM, which is achieved by means of the 1451.1 API and the IO T-Block API previously shown in Figure 3-2.

The application layer shall also act as a bridge between the network operations and the transducer operations. It shall have application-specific code that is pertinent to the particular control or monitoring system that is implemented.

3.2.2 Network Layer

The network layer has hardware and software blocks that handle the communications with the underlying network. The hardware consists of the logic blocks needed for the network communication protocol. The requirement for the hardware is that it meets the electrical and logical specifications of the underlying network as well as those of the Excalibur chip. The software blocks consist of the network library which is network-specific and the 1451.1 API defined by the NCAP Block of the standard's object model.

The network library consists of the “driver” that shall be responsible for encoding and decoding data to/from the on-the-wire format of the underlying network. The 1451.1 API shall provide the network independency of the design, since it shall hide the communication details of the network driver.

The term IEEE 1451.1 Network Communication is defined by the standard as the communication between two objects in distinct process spaces either within a single NCAP or over the network between the two NCAPs.

Therefore, to prove that the system supports network communications either of the communications stated before shall be executed. Those communications shall be made using the two communication models that are defined in the standard (client-server and/or publish-subscribe).

3.2.3 Transducer Layer

The transducer layer contains both hardware and software blocks that handle the transducer-side communications of the NCAP. The hardware block shall be responsible for the physical

connection between the NCAP and TIM. This block shall meet the electrical and timing specifications of the Excalibur chip.

The software shall be responsible for interactions with any transducer that is physically connected to the NCAP. This shall be done through the IO T-Block API defined in the standard by the Transducer Block class. Therefore, this block shall be responsible for decoding and encoding information that is sent and received by the NCAP/TIM communication. For example, if the application needs to send a trigger command to the TIM, it signals this to the API. The API then puts the command in the format of the NCAP/TIM communication protocol and sends it.

All the commands that are supported by the TIM shall be represented within this layer. A common command that shall be represented in the transducer layer is the trigger command. This command allows the NCAP to read/set the transducers that are physically connected to the system. There are different steps that shall be followed when reading or setting an actuator. For example, the steps for reading a sensor are as follows:

1. Select the channel of the sensor that will be used.
2. Trigger the Sensor.
3. Wait until the TIM indicated a reading is available (TIM ACK).
4. Access the raw sensor reading.
5. Convert raw sensor reading into SI units using the information stored in the TEDS.

Similarly, the sequence for setting an actuator shall be the following:

1. Select the actuator channel.
2. Convert the SI units into a raw actuator setting from the information stored in the TEDS.

3. Write the raw actuator setting.
4. Trigger the Actuator.
5. Wait until the TIM indicates that the action is complete (TIM ACK).

There shall also be the possibility of reading/setting all the transducers that are implemented in the system. The sequence for this operation (global trigger) is complicated and dependent upon the types of transducers that are in the application. For a system with sensors and actuators, it shall be the programmer's responsibility to make sure that each actuator receives a different data set.

3.3 TRANSDUCER INTERFACE MODULE

The Transducer Interface Module (TIM) block contains a combination of hardware and software blocks that shall maintain the functionality of a smart transducer as is stated in the IEEE 1451 family of standards. In order for the TIM to be deemed "smart", it shall have the capability of self-identification, which is achieved by the TEDS. Other functionality that the TIM shall have includes the ability to communicate with an NCAP, handle triggering, generate interrupts, and interface with the physical transducers.

The TIM shall be initialized (by the NCAP or by itself) after it is powered-on, and then shall enter the operational state until it is reset by the NCAP or is powered down.

3.3.1 Transducer Electronic Data Sheet

The Transducer Electronic Data Sheet (TEDS) is at the core of the TIM's functionality because it provides the idea of self-identifying transducers, which enables the idea of "plug and play". This block may be generated either at manufacture time or remotely via an NCAP.

The TEDS shall reside in non-volatile memory and completely describe the TIM (the entity and its transducers). There shall be a block labeled Meta-TEDS that describes the TIM in its entirety. There shall also be a block labeled Channel-TEDS that describes each implemented transducer. Other blocks that have information such as calibration are not mandatory and will not be implemented in this design.

The information that shall be represented by the Meta-TEDS includes the amount of transducers that are implemented, the maximum and minimum sampling rate of the system, and a checksum for data integrity.

The Channel-TEDS information shall include the type of transducer (actuator, sensor), the physical units of the transducer (e.g. Temperature in Celsius), the data model, timing information such as update time, write setup time, sampling period, etc. and a checksum for data integrity.

3.3.2 NCAP Communication

The TIM shall communicate with an NCAP using a protocol that will take advantage of the lack of physical separation between the two modules. The structure of this connection may resemble the STIM/NCAP connection from the 1451.2 standard since there may only be one TIM connected to an NCAP. Unlike what is allowed by the 1451.3 standard, in which multiple

Transducer Bus Interface Modules (TBIM) can connect to a single NCAP. This interface shall support the different commands that can be sent to the TIM, and shall also provide the capability of interrupts over the protocol as well as access to the different modules in the TIM.

The set of mandatory commands that shall be fully implemented are Read TEDS, Read/Write Interrupt masks, Read Status, Trigger, and Read/Write Transducer Data. These commands shall be applied to either an individual implemented channel or to the TIM as a whole (CHANNEL_ZERO).

3.3.3 Trigger

The TIM shall handle both individual and global triggering. Figure 3-3 shows use-case diagrams depicting the behavior of the TIM when it receives either an individual or global trigger command.

When an individual trigger is sent by the NCAP, the TIM shall decode the command to select the channel for which it is intended and set the appropriate signals to sample/set the transducer. Then, the TIM shall generate an acknowledge signal (TIM ACK) when the sensor is completely sampled (ADC finished conversion, etc) or the actuator has completely acquired a data set. The signal tells the NCAP that the command was executed and that there is information that it should read.

When a global trigger occurs, the TIM shall sample/set all the transducers in the system. The requirement is that each individual channel generates an ACK signal when it executes the trigger command successfully. CHANNEL_ZERO shall generate the ACK signal when every channel in the system has been completely sampled/set.

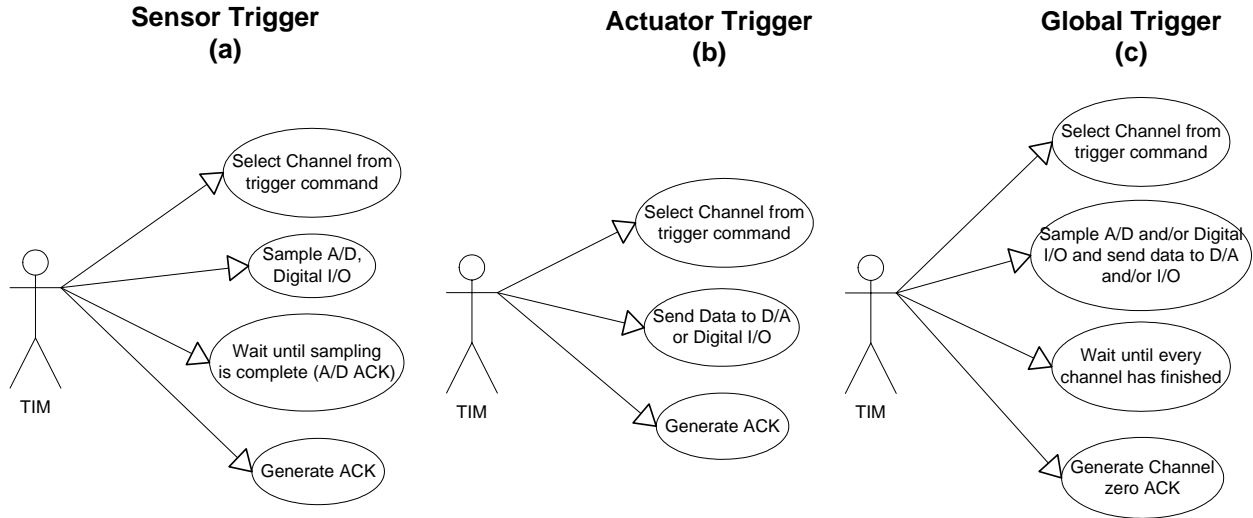


Figure 3-3 TIM trigger behavior
(a) Sensor Trigger, (b) Actuator Trigger, (c) Global Trigger

3.3.4 Status and Interrupts

The status of the TIM and its individual channels shall be represented in such a way that it is accessible by the NCAP. The status shall include information regarding the operation of the device. Table 3-1 summarizes the different status bits that shall be represented in the TIM.

Table 3-1 TIM's Status Bits

TIM CHANNEL_ZERO	Individual Channel
CHANNEL_ZERO Trigger Acknowledged	Channel Trigger Acknowledged
Invalid Command	Reserved
TIM Operational	Channel Operational
Corrections enabled/disabled	Corrections enabled/disabled

The *TIM Channel Trigger Acknowledged* bit shall be set by the TIM when it acknowledges the trigger signal and shall be cleared when read. After global triggers, every channel trigger acknowledge bit shall be set by the TIM when it would have acknowledged each channel trigger if they were individually addressed.

The *TIM trigger acknowledged* bit shall be set by the TIM when every implemented channel acknowledges their respective triggers.

The *invalid command* bit shall be set by the TIM when the NCAP sends a command that is not implemented in the TIM. It shall be cleared when read or if the condition goes away.

The *TIM operational* bit shall be set by the TIM after the module is completely initialized.

The *channel operational* bit shall be set by the TIM after the TIM and the particular channel have been completely initialized.

The *corrections enabled/disabled* bit in both the CHANNEL_ZERO and Individual Channel definition shall be set if the TIM has the capability apply corrections to the transducer data.

The TIM shall generate interrupts by a combination of the status and interrupt mask bits. The status and interrupt bits shall be compared on a one to one (bit by bit) basis to determine if an interrupt exists.

The default setting for the interrupt mask shall be all ones (every status bit can generate an interrupt). It shall be the responsibility of the NCAP to service the interrupt as well as setting/clearing the interrupt mask.

3.4 PHYSICAL WORLD

The transducers are not defined in the standard giving the designer the freedom to decide what is best for the particular application that will be built. For transducers, it is usually the case that

they are implemented off-chip and interface to the TIM by means of an Analog to Digital Converter (ADC), Digital to Analog Converter (DAC) or digital IO. For this implementation, transducers and their respective data converters will be implemented off chip. These off-chip blocks shall meet the electrical specifications of the Excalibur chip and they shall also comply with the particular timing specifications of the data converters and/or digital IO.

4.0 SPECIFICATIONS

The specifications for this IEEE 1451 implementation are independent of the network employed and the transducers that are interfaced to the application system. The NCAP's behavior is completely described using the specifications of the IEEE 1451.1 standard, while the TIM will have the functionality that a smart transducer is required to have as is stated by the standards.

4.1 NETWORK CAPABLE APPLICATION PROCESSOR

The information model of the IEEE 1451.1 standard defines three different models for the NCAP that enable network and transducer independency. These models include, an object model; this model defines transducer device specific abstract objects – or, classes with attributes, methods, and state behavior. A data model; this model defines information encoding rules for transmitting information across both local and remote object interfaces. A network communication model; this model supports a client/server and publish/subscribe for communicating information between NCAPs.

Therefore, by designing and implementing the different objects that are defined in the information model that is defined in the standard, we will achieve network and transducer independency. This is because the standard-defined objects hide the communication details of

the NCAP with a particular network and transducers, making the application's behavior transparent to the physical connections of the NCAP.

In order to implement the information model, there are different properties among the NCAP's objects that need to be taken into consideration. These properties include a class hierarchy and an owning relationship that can be described using Unified Modeling Language (UML) ⁽²¹⁾. These properties are shown in Figure 4-1.

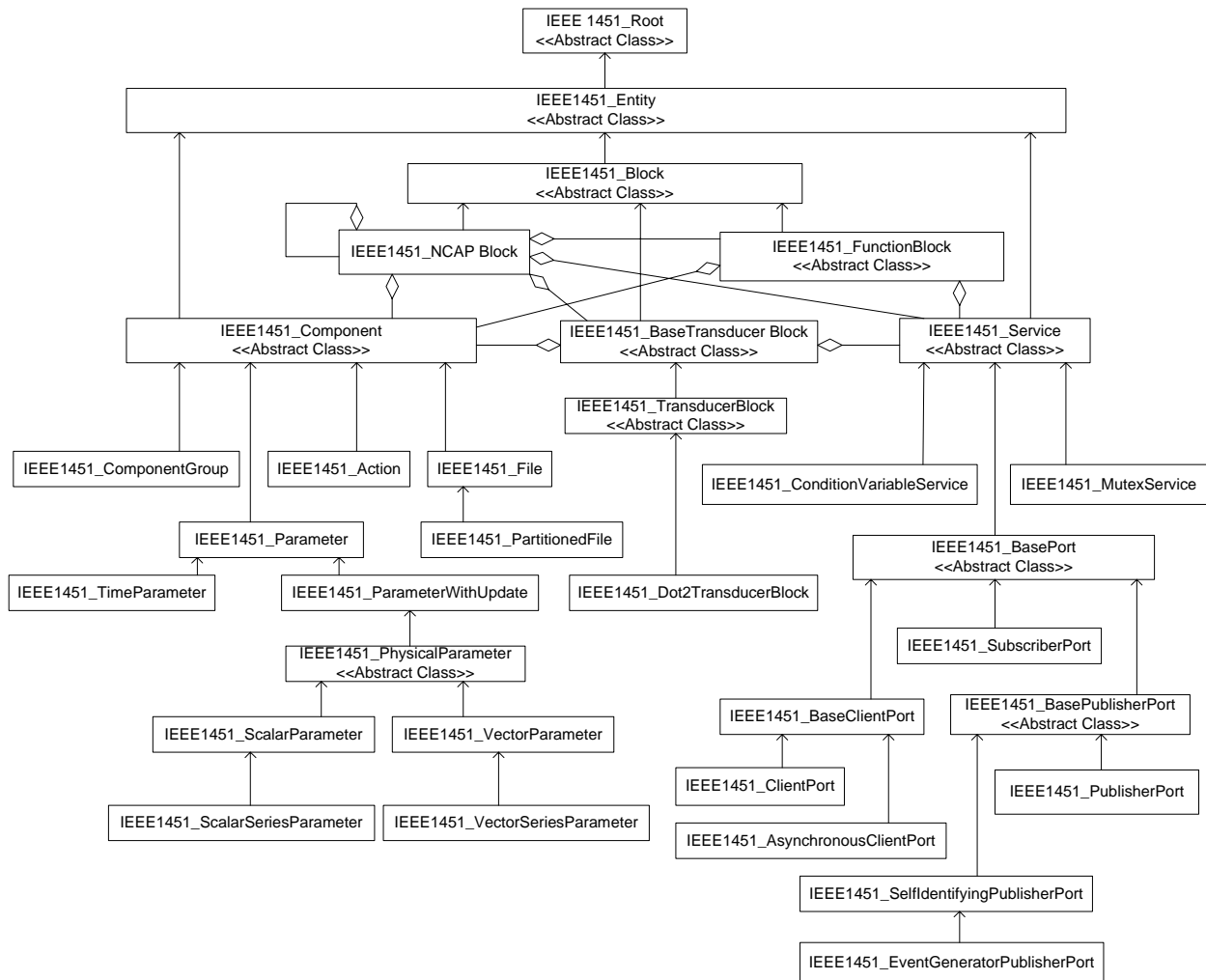


Figure 4-1 UML Model of IEEE 1451.1 Objects

The figure shows the inter-object relationships (inheritance and ownership) through the use of two types of arrows. The diamond-based arrows are used to denote the owning

relationship between the objects by pointing to the block that owns the other block. For example, the NCAP Block owns the Component block, the Function Block, and so forth. Note that only Block Objects can own other classes.

The normal-shaped arrows denote the inheritance relationship between the child and the parent classes. For example, IEEE1451_Root class is the parent of IEEE1451_Entity who inherits all the functionality of its parent class.

The next step is to define the classes shown in the figure. These classes will be implemented following the specifications of the 1451.1 standard. The different object classes that are defined in the standard can be further explained as follows:

- The *Root class* is the origin for the class hierarchy of all objects that are defined in the standard.
- The *Entity abstract class* is the root for the class hierarchy of all objects defined by this standard that may be made visible over the network.
- The *Block abstract class* is the root for the class hierarchy of all Block objects.
- The *NCAP Block class* provides resources and operations within an NCAP process to support Block, Service, and Component management. This support includes registration, deregistration, initialization and startup, and shutdown.
- The *Function Block class* is the root for the class hierarchy of all Function Block objects. The Function Block is the primary mechanism for the abstraction and packaging of application functionality.
- The *Base Transducer Block class* is the root for the class hierarchy of all Transducer Block objects.

- The *Transducer Block class* is the root for the class hierarchy of all Transducer Block objects in the family of transducers specified by IEEE 1451.X standards.
- The *Component abstract class* is the root for the class hierarchy of all Component objects.
- The *Parameter class* is used to model network visible variables and to provide a means for accessing them.
- The *Parameter With Update class* is used to model network visible variables, and to provide a means for accessing the variable. This class has an associated mechanism that supports an update action involving the variable.
- The *Physical Parameter abstract class* and its subclasses are used to represent network visible variables, modeled by the Parameter With Update class that directly or indirectly represent the physical world. The Physical Parameter provides the information necessary to interpret a measurement or actuation.
- The *Scalar Parameter class* is used to model physical world quantities that do not have dimensions or orientations associated with them, and are appropriately represented as mathematical scalars.
- The *Scalar Series Parameter class* is used to model physical world quantities, best modeled as a succession of scalars evenly distributed along some dimension.
- The *Vector Parameter class* is used to model physical world quantities that have multiple dimensions and perhaps orientation associated with them, and is appropriately represented as mathematical vectors.

- The *Vector Series Parameter class* is used to model a uniform series of physical world quantities that have dimensions and orientation associated with them and is appropriately represented as mathematical vectors.
- The *Time Parameter class* is used to represent time parametric values. The purpose of this class and its subclasses is to model network visible variables that directly or indirectly represent the time of some event, or the duration between two events, where the significant characteristic of the event is the time rather than some other value.
- The *Action class* provides a model to represent activities that alter the system state and that require significant time to execute compared to other activities in the system.
- The *File class* is an abstraction of a data resource. Files represent a block of memory, which may be opened, closed, read from, and written to.
- The *Partitioned File class* is used for files that are subdivided into a number of partitions.
- The *Component Group class* provides a way to specify set membership relations between objects in a system.
- The *Service abstract class* shall be the root for the class hierarchy of all Service objects. The Service classes represent object types used to support communication and other aspects of block functionality.
- The *Base Port abstract class* is the root for the class hierarchy of all communication port objects used to send communications via the underlying network.

- The *Base Client Port abstract class* is the root for the class hierarchy of all client-server communication client-side port objects.
- The *Client Port class* provides the client-side application interface to client-server communications. This class abstracts the details of the specific network. Two models of client-server communication are provided: blocking and “send and forget”.
- The *Asynchronous Client Port class* provides the client-side functionality for an asynchronous, non-blocking, client-server communication model.
- The *Base Publisher Port* provides basic publisher-side functionality for its subclasses.
- The *Publisher Port class* provides publisher-side functionality for a publish-subscribe communication model.
- The *Self Identifying Publisher Port class* provides publisher-side functionality for a publish-subscribe communication model with operations to allow the subscriber to establish communication with the publisher, and the publisher to notify subscribers of changes in the publication policy.
- The *Event Generator Publisher Port class* is used to allow events internal to the operation of a block to result in the publication of an event record.
- The *Subscriber Port class* provides objects with a mechanism for subscribing to publications.
- The *Mutex Service class* provides mutual exclusion capability.
- The *Condition Variable Service class* provides the capability for ordering concurrent activities.

In order to identify the different objects previously mentioned, we will use the identifying properties that are defined in the standard. Therefore, we will identify the object by its *Class ID*, *Class Name*, *Object ID*, *Object Tag*, *Object Name*, and *Object Dispatch Address*.

The *Class ID* identifies the object's class, the position in the class hierarchy, and cannot be modified.

The *Class Name* provides a human-readable description of the semantics of the class and it cannot be modified.

The *Object ID* is unique within a system and it unambiguously distinguishes the object from any other object.

The *Object Tag* is unique within a system and usually defines a logical endpoint for the server side of client-server communications.

The *Object Name* provides a human-readable description of the semantics of an instance of a class and it will be bound when the object is created.

The *Object Dispatch Address* is the network-specific address used by the underlying network infrastructure to address the object. For example, in an Ethernet network this value would be the IP address.

The rest of this section goes into detail about the complete information model. This includes the data model (for the encoding rules for transmitting information across both local and remote object interfaces), a functional overview (high-level object interaction in the NCAP), and the top-level class definition (for the objects that hide the communication details of network and transducer communications).

4.1.1 Data Model

The data model of the standard defines a variety of datatypes (primitive and derived) that will be used for the functionality of the object classes. The primitive datatypes will be mapped to the programming language that will be used. The definitions for these types are shown in Table 4-1. The derived datatypes will be derived from these primitive types. These types are used to represent structures.

Table 4-1 Simple Primitive Types

Datatype	Default Value	Definition
Boolean	FALSE	TRUE or FALSE
Integer8	0	8-bit signed integer
UInteger8	0	8-bit unsigned integer
Integer16	0	16-bit signed integer
UInteger16	0	16-bit unsigned integer
Integer32	0	32-bit signed integer
UInteger32	0	32-bit unsigned integer
Integer64	0	64-bit signed integer
UInteger64	0	64-bit unsigned integer
Float32	+0.0	IEEE Std 754-1985 single-precision floating point number
Float64	+0.0	IEEE Std 754-1985 double-precision floating point number
Octet	All Bits set to 0	8-bit quantity not interpreted as a number

Next, we describe the IEEE 1451 *String* datatype which is represented by a structure. This type contains four fields that are used to represent the character set, character code, language, and the string data. The first three mentioned fields are represented as 8-bit unsigned integers. These fields have an enumeration associated with them that is used to represent different languages, character sets, and character codes. The last field in the structure (string data) is represented as an octet array. The size of the array is set by the enumeration for the character code. Also, to interpret the string data there is an enumeration associated for each of the supported languages in the NCAP implementation.

An important derived datatype that will be used in communications with networks is the *Argument* datatype. This type is a container, which can hold any of the other 1451.1 types. All application data in a network communication will be carried in arrays of *Argument*, hence using the *Argument Array* datatype.

4.1.1.1 Class Header Format and Return Codes

The class header format will be used for all the object classes that were previously shown. The format of the header is shown in Table 4-2.

Table 4-2 Class Header Format

Format	Description
Class ID	It is represented as an array of an Octet type (8-bit unsigned char), and is used to encode the position of the class in the class hierarchy.
Class Description	It is the formal name for the class interpreted as an IEEE 1451 String.
Parent Class Name	It is the value of the class descriptor of the immediate parent class in the hierarchy from which this class is sub-classed.

The different operations that are defined for the object classes have two return types, *OpReturnCode* and *ClientServerReturnCode* (used for client-server communications).

The *OpReturnCode* is a 16 bits unsigned integer that is expressed as a sequence of two fields (minor and major) and is used as the return type for most IEEE 1451.1 operations. The minor field is the high order 8 bits while the major field is represented by the low-order 8 bits. For example, an *OpReturnCode* value 0x103 (HEX notation) would have a major field of 3 and a Minor field of 1. The combination of these fields is used to determine the result of the operation.

The *ClientServerReturnCode* is a 32 bit unsigned integer interpreted as a sequence of four fields (portCode, performCode, operationMinorCode, and operationMajorCode). The portCode is the return code for the client-side function and it is the high order 8 bits of the code. The performCode is the return code for the server-side operation. Lastly the minor and major codes are used to show the *OpReturnCode* of these operations. For example if the *ClientServerReturnCode* is 0x04020103 (HEX notation), the field values would be as follows: portCode = 4, performCode = 2, operationMinorCode = 1, and operationMajorCode = 3.

4.1.2 Functional Overview and Top-level class definitions

For the object interaction, the standard uses a “backplane” or “card cage” concept for the different objects that can be *plugged* to the NCAP, in a similar fashion to what is done in the USB 1.1 or 2.0 standard and other “plug and play” devices. Block classes form the major blocks of functionality that can be plugged into the card-cage to create various types of devices. This relationship is shown in Figure 4-2.

Note that the NCAP Block centralizes and “glues” all the system and communications facilities together. Network communications are viewed as ports, Function block application code is *plugged* in as needed, and Transducer blocks map the physical transducer to the NCAP. The following sub-sections will go into detail about these classes. Note that we will only describe the objects that are directly used in network and transducer communications. For a detailed description about the rest of the classes and their functionality refer to Appendix A.

To describe the structure and functionality for the object classes we will use UML diagrams. In doing so, we define the class header as the attributes of the class, network visible operations as public, and local operations as protected.

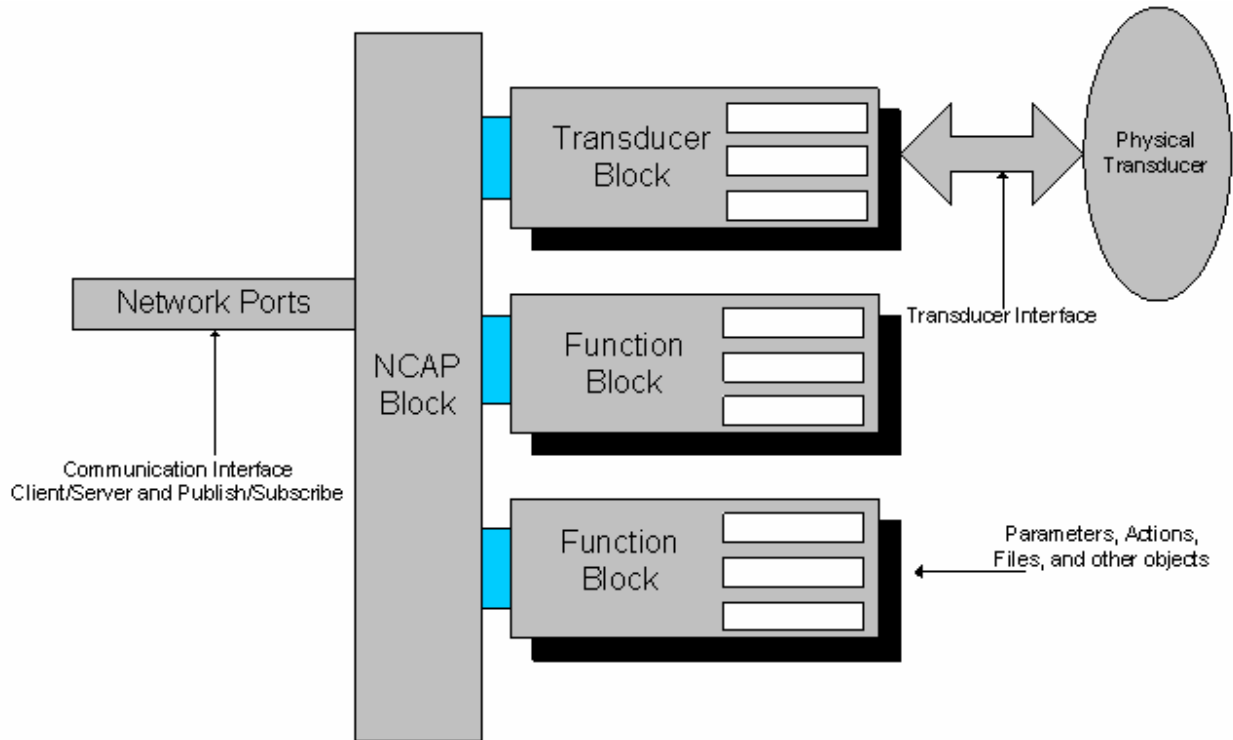


Figure 4-2 Conceptual View of an IEEE 1451.1 NCAP

4.1.2.1 Block class

We start with the Block class since it is the root hierarchy for all block objects and block objects are the core of the APIs defined for the NCAP. The UML class diagram for the Block class is shown in Figure 4-3.

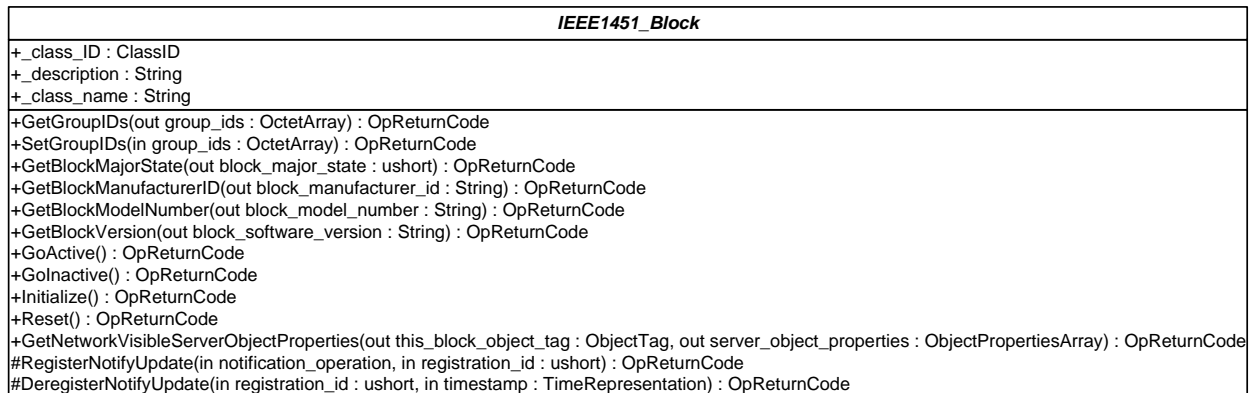


Figure 4-3 Block Class UML Diagram

The different operations that are described for this class are as follows:

- The *GetGroupIDs* operation will be used obtain the identifiers that represent sets of objects of which this Block instance is a member.
- The *SetGroupIDs* operation will be used to initialize/modify the identifiers that represent sets of objects of which this Block instance is a member. For example, all Blocks in a particular control group could be given a common group ids value.
- The *GetBlockMajorState* operation will be used to obtain the current state of the state machine that controls this block's behavior.
- The *GetBlockManufacturerID* operation will be used to obtain a parameter that identifies the manufacturer of the Block. For our implementation this operation will return a zero as the manufacturer ID.
- The *GetBlockModelNumber* operation will be used to obtain an identifier (represented as a 1451.1 String) that is used to distinguish between different implementations of the class.
- The *GetBlockVersion* operation will be used to obtain an identifier (represented as a 1451.1 String) that is used to distinguish between different implementations of the model.
- The *GetNetworkVisibleServerObjectProperties* operation will be used to obtain the *ObjectTag* of the target Block that is executing and the server object properties. The server object properties datatype consists of a data structure that contains information regarding the server object.
- The *GoActive* operation will be used for transitions within the state machine for this block. The transition will be from the uninitialized state to the inactive state.

- The *GoInactive* operation will be used for transitions within the state machine for this block. The transition will be from the active state to the inactive state.
- The *Initialize* operation will be used for transitions within the state machine for this block. This transition will be from the uninitialized to the initialized state.
- The *Reset* operation will be used for transitions within the state machine for this block. This transition will be from any state back to the uninitialized state.
- The *RegisterNotifyOnUpdate* and *DeRegisterNotifyOnUpdate* operations are defined as optional in the standard and will not be implemented in this design.

The behavior for this class is controlled by a state machine with three states. This state machine is shown in Figure 4-4. The BL_UNINITIALIZED state is reserved for local activities related to bringing the Block Object into existence and performing any related local preparations needed for the Block function. While in this state any classes owned by the Block class cannot execute any network communications, but it is in this state where owned objects are registered with the Block for network communications.

The BL_INACTIVE state is reserved for activities such as the configuration of the network communication properties of the Block and its owned objects, initialization, and diagnosis and maintenance of the Block object. The BL_ACTIVE state is reserved for activities related to the normal application function of the block.

Every Block object defined in the standard will inherit the state machine shown in Figure 4-4, as well as the network visible operations that are shown in Figure 4-3.

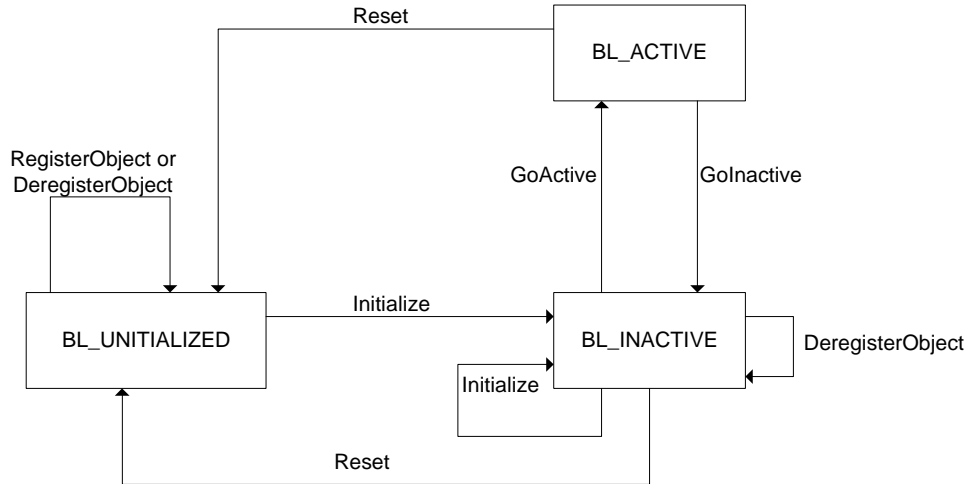


Figure 4-4 State Machine for Block Class

4.1.2.2 NCAP Block Class

The next step is to define the NCAP Block class since it is the key source for network communications and system configuration as was previously discussed in Section 3.2. It is important to note that this object class owns every other Block object within the same hierarchy and it is the only one to own itself as was previously shown in Figure 4-1.

This class is the key source for system configuration and bookkeeping information about its Network Visible Owned Objects. The structure for this class is shown in Figure 4-5.

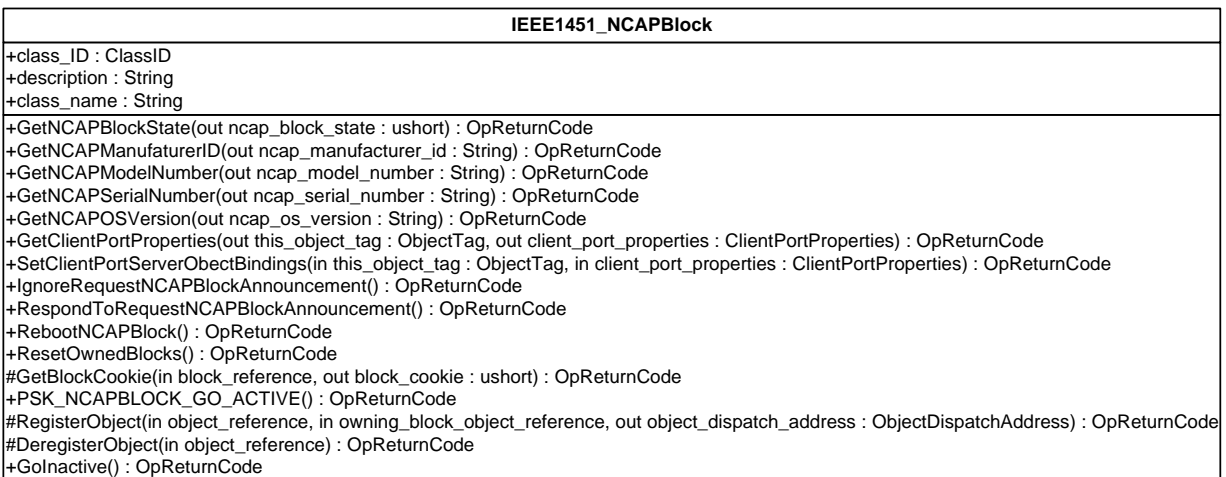


Figure 4-5 NCAP Block UML Class Diagram

The following discussion goes into detail about this class' operations and their top-level functionality.

- The *GetNCAPBlockState* operation will be used to obtain the current state of the state machine for this object.
- The *GetNCAPManufacturerID* operation will be used to identify the manufacturer of the NCAP.
- The *GetNCAPModelNumber* operation will return an identifier that is used to distinguish between different NCAP implementations.
- The *GetNCAPSerialNumber* operation will be used to distinguish different instances of NCAP implementations.
- The *GetNCAPOSVersion* operation will return an identifier assigned by the manufacturer to specify the operating system that is in use.
- The *GetClientPortProperties* will be used to obtain the *ObjectTag* of the NCAP Block as well as client port information.
- The *SetClientPortPropertiesBindings* will be used to initialize or modify the *ObjectTag* of the NCAP Block as well as the client port information.
- The *IgnoreRequestNCAPBlockAnnouncement* operation will be used to ignore a publication that provides a notification of the existence of an NCAP Block object within the system.
- The *RespondToRequestNCAPBlockAnnouncement* operation is used to respond to a publication that provides a notification of the existence of an NCAP Block object within the system.

- The *RebootNCAPBlock* operation will be used to place the NCAP Block and all its owned objects to be placed in their default power-on state.
- The *ResetOwnedBlocks* operation will cause all objects owned by the NCAP Block class to behave as if they just received a reset operation.
- The *GetBlockCookie* operation will be used to obtain the block cookie (used to tell a client whether the context of the server has changed) of the particular object that is being accessed.
- The *PSK_NCAPBLOCK_GO_ACTIVE* operation will be used for transitions within the state machine for this block. This transition will from the active to the initialized state.
- The *RegisterObject* and *DeRegisterObject* operations are optional and will not be implemented.
- The *GoInactive* operation will be used for transitions within the state machine for this block. This transition will be from the active to the initialized state.

The behavior of this class is controlled by the state machine for the Block object that was previously shown in Figure 4-4. However, the NCAP Block sub-states the BL_INACTIVE state to include two states NB_INITIALIZED and NB_ERROR. The sub-states are shown in Figure 4-6.

The initial transition from the BL_UNINITIALIZED state to the NB_INITIALIZED state will be caused by implementation-specific mechanisms within the NCAP Block. For this implementation, this transition will occur when the *Initialize* operation of the Block class is called.

The “Fail” signal shown in the state machine is an internally generated transition that causes the NCAP to go from the NB_INITIALIZED to the NB_ERROR sub-state. The *GoInactive* operation causes the transition to the NB_ERROR sub-state if the NCAP detects an error.

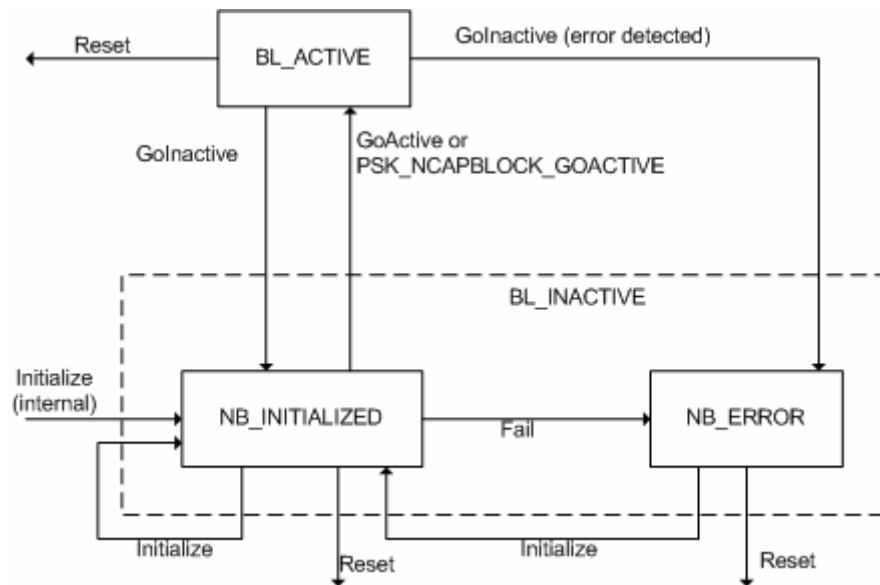


Figure 4-6 State Machine for the NCAP Block

4.1.2.3 Function Block Class

The Function Block class (shown in Figure 4-7) will be used as the primary mechanism for the abstraction and packaging of application functionality. Therefore, the application-specific objects will be owned and controlled by the Function block. Similarly, any interaction between the application’s objects and other standard-defined objects will be done through the use of this class.

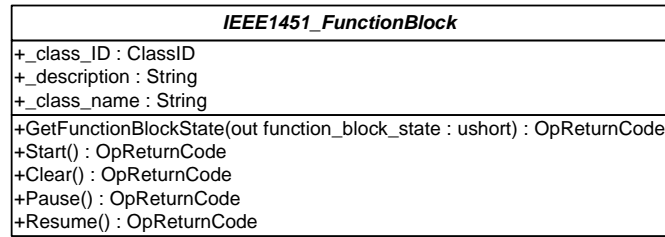


Figure 4-7 Function Block UML Class Diagram

The definitions for the operations for this class are as follows:

- The *GetFunctionBlockState* operation will be used to obtain the current state of the state machine for this object.
- The *Start* operation will be used for transitions within the state machine for this block. This transition will be from the idle to the running state.
- The *Clear* operation will be used for transitions within the state machine for this block. This transition will be from the running to the idle state.
- The *Pause* operation will be used for transitions within the state machine for this block. This transition will be from the running to the stopped state.
- The *Resume* operation will be used for transitions within the state machine for this block. This transition will be from the stopped to the running state.

This block's behavior is controlled by the inherited state machine of the Block class that was shown in Figure 4-4. However, the BL_ACTIVE state is sub-stated as is shown in Figure 4-8.

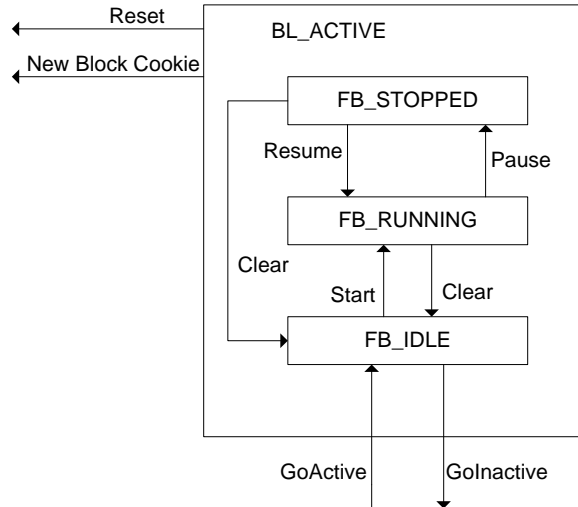


Figure 4-8 State Machine for a Function Block

4.1.2.4 Client-Server Network Communication Classes

There are two communication models that will be supported by this NCAP. These models are the client-server and publish-subscribe models. In this section, we will focus on the standard-defined software blocks that enable the NCAP to engage in network communications.

We start with the client-server model. This model is a tightly coupled communication model used for one-to-one communication. The server-side of the NCAP is provided by Entity abstract class (shown in Figure 4-9) by means of the *Perform* operation.

<i>IEEE1451_Entity</i>
+_class_ID : ClassID +_description : String +_class_name : String
+GetObjectTag(out object_tag : ObjectTag) : OpReturnCode +SetObjectTag(in object_tag : ObjectTag) : OpReturnCode +GetObjectID(out object_id : ObjectID) : OpReturnCode +GetObjectName(out object_name : String) : OpReturnCode +GetDispatchAddress(out dispatch_address : ObjectDispatchAddress) : OpReturnCode +GetOwningBlockObecjtTag(out owning_block_object_tag : ObjectTag) : OpReturnCode +GetObjectProperties(out object_properties : ObjectProperties) : OpReturnCode #Perform(in server_operation_id : ushort, in server_input_arguments : ArgumentArray, out server_output_arguments : ArgumentArray) : ClientServerReturnCode

Figure 4-9 Entity UML Class Diagram

The operations that are defined for the Entity class can be further explained as follows:

- The *GetObjectTag* operation will be used to obtain the *ObjectTag* of the object that is being accessed.
- The *SetObjectTag* operation will be used to initialize or modify the *ObjectTag* of the object that is being accessed.
- The *GetObjectID* operation will be used to obtain the current value of the object ID for the object that is being accessed.
- The *GetObjectName* operation will be used to obtain the current value of the object name for the object that is being accessed.
- The *GetDispatchAddress* operation will be used to obtain the value of the object dispatch address of the object.
- The *GetOwningBlockObjectTag* operation will be used to return the *ObjectTag* of the owning block object.
- The *GetObjectProperties* operation will be used to obtain the current values of the *ObjectTag*, dispatch address, object name, the associated block cookie of this object, and the *ObjectTag* of the owning block of this object.
- The *Perform* operation will be used as the server-side construct for client-server communications.

On the other hand, the client-side functionality is given by the Client Port class (shown in Figure 4-10) through the *Execute* function.

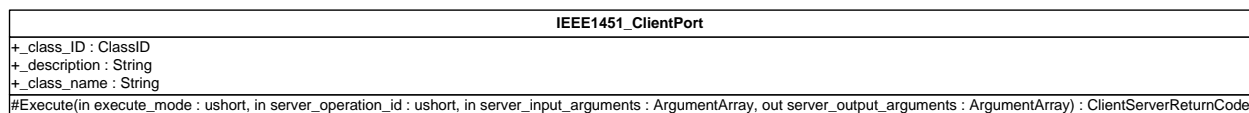


Figure 4-10 Client Port UML Class Diagram

Now that we have shown the client-server operations and their structures, we can describe their functionality. First, we begin by showing the system configuration for this communication model.

The network port information will be set at compile-time as was discussed earlier in this Section, so there will be no dynamic system configuration for the network communications. Therefore, as soon as the NCAP is operational the server and client objects will be able to communicate across the underlying network.

Next, we can describe the complete interaction for a client-server operation. This interaction is shown in Figure 4-11. Note that the first thing that will be done is to encode the arguments that will get sent over the network. After this, the Client Port invokes the *Execute* operation which will have the server information as well as the packet that will be sent over the network.

This information will then be sent to the network infrastructure so that it can be marshaled onto the on-the-wire format of the particular network that is employed. After this information is sent over the network, the network infrastructure for the server-side will de-marshal the information and call the *Perform* operation so that the information can be decoded. Then, the operation can be executed and the results can be sent back to the client-side so that it knows that the operation completed successfully.

It is important to note that the client-server interaction does not necessarily need to return to the client as is shown in Figure 4-11, since there are two execution modes for the *Execute* operation. If the execution mode is EM_NO_RETURN_VALUE then the client object sends information over the network and does not expect a return value (send and forget) from the server-side. The figure assumes the blocking execution mode labeled EM_RETURN_VALUE.

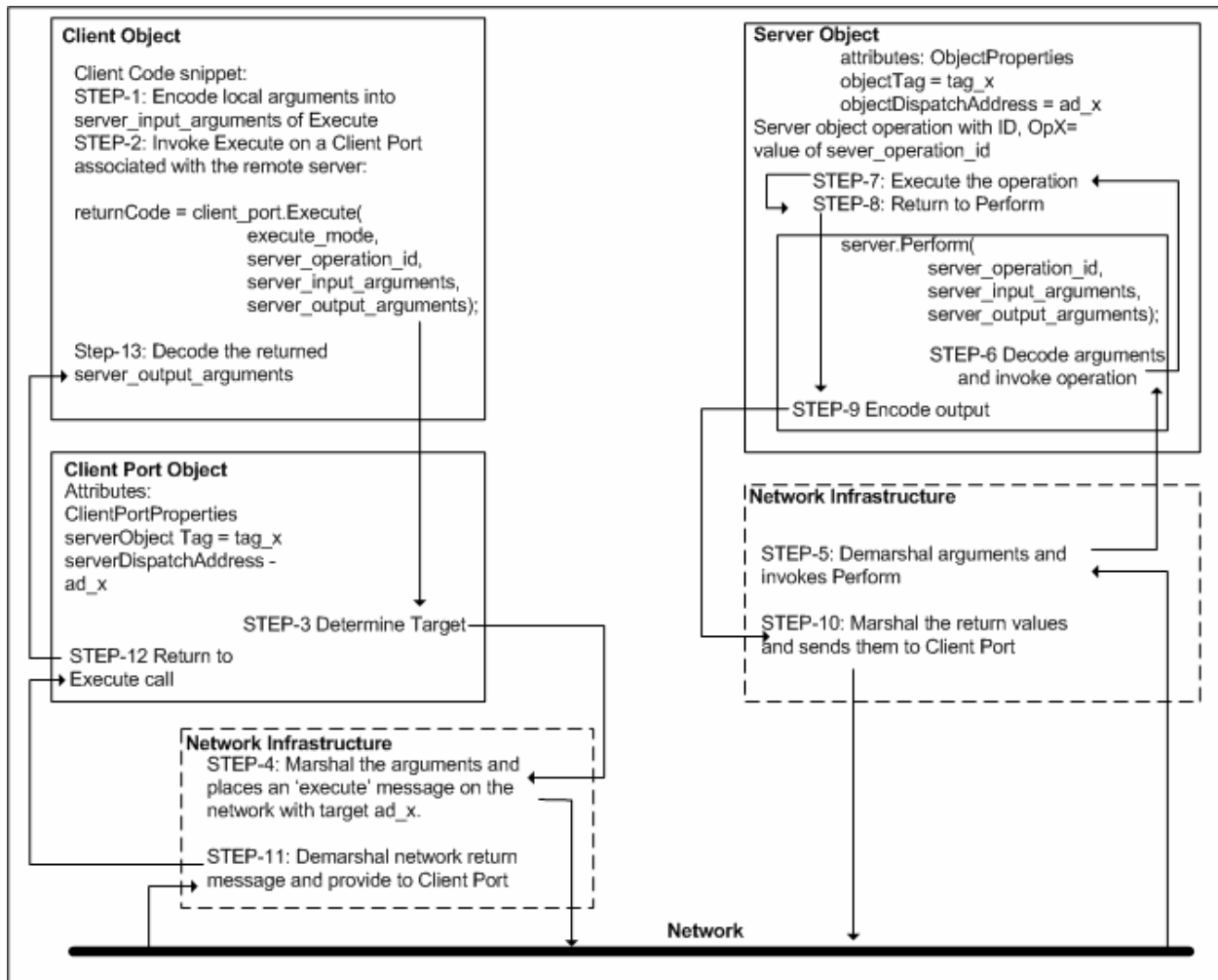


Figure 4-11 Client-Server Communication Model

4.1.2.5 Publish-Subscribe Network Communication Classes

The publish-subscribe model is a loosely coupled communication model for one-to-many or many-to-many communications. The Publisher Port class (shown in Figure 4-12) provides the publisher-side functionality through the *Publish* operation.

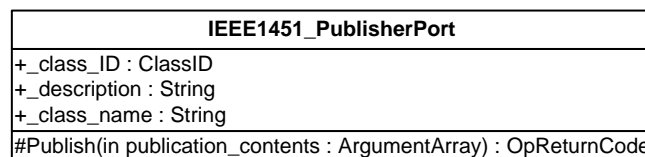


Figure 4-12 Publisher Port UML Class Diagram

On the other hand, the Subscriber Port class provides objects with a mechanism for subscribing to publications. The structure for this class is shown in Figure 4-13.

IEEE1451_SubscriberPort
+_class_ID : ClassID +_description : String +_class_name : String
+SetSubscriptionQualifier(in subscription_qualifier : SubscriptionQualifier) : OpReturnCode +GetSubscriptionQualifier(out subscription_qualifier : SubscriptionQualifier) : OpReturnCode +GetSubscriptionKey(out subscription_key : ushort) : OpReturnCode +SetSubscriptionDomain(in subscription_domain : PubSubDomain) : OpReturnCode +GetSubscriptionDomain(out subscription_domain : PubSubDomain) : OpReturnCode #AddSubscriber(in notification_operation_reference, in subscription_id : ushort) : OpReturnCode #DeleteSubscriber(in notification_operation_reference, in subscription_id : ushort) : OpReturnCode

Figure 4-13 Subscriber Port UML Class Diagram

The operations that are defined for the Subscriber Port class can be summarized as follows:

- The *SetSubscriptionQualifier* operation will be used to initialize or modify the current value of the port's subscription qualifier. The subscription qualifier is used to determine which publications will be accepted by the port.
- The *GetSubscriptionQualifier* operation will be used to obtain the current value of the port's subscription qualifier.
- The *GetSubscriptionKey* operation will be used to obtain the current value of the port's subscription key. This key is used to determine the type of publication that the port is subscribing to.
- The *GetSubscriptionDomain* operation will be used to obtain the current value of the subscription domain defining a ser of candidate publications to be accepted by the port.
- The *AddSubscriber* and *DeleteSubscriber* operations are optional and will not be implemented.

The operations previously defined will be used for publish-subscribe network communications. The interaction between the operations and objects can be further explained as follows:

- The Publisher object invokes the *Publish* operation on its associated local Publisher Port passing in as an input argument the publication's contents.
- Using the network infrastructure, the invocation of the *Publish* operation results in the delivery of the publication to all Subscriber Ports in the publication's Domain.
- The receiving Subscriber Ports each use the values of their Subscription Key, Subscription Domain, and Subscription Qualifier attributes to filter the incoming publication.
- If the publication passes a Subscriber Port's filter, the Port invokes all of its registered Subscribers' callback operations, providing as an input argument the publication's contents. Subscribers will be registered at compile-time.

4.1.2.6 Transducer Block Class

This class establishes the mapping between the individual channels of the TIM transducers and the public transducers of the Transducer Block in the NCAP. The most relevant operations that will be done through this class are the access of the Meta-TEDS as well as global triggering. The structure for this class is shown in Figure 4-14.

<i>IEEE1451_TransducerBlock</i>
+_classID : ClassID +_description : String +_class_name : String
+GetCorrectionMode(out correction_mode : ushort) : OpReturnCode +GetNumberOfTransducerChannels(out number_of_transducer_channels : ushort) : OpReturnCode +GetMinimumSamplingPeriod(out minimum_sampling_period : TimeRepresentation) : OpReturnCode +GetChannelParameterObjectChannelNumbers(in channel_number : ushort, out parameter_object_tags : ObjectTag) : OpReturnCode +GetUnrepresentedChannelNumber(out unrepresented_channel_numbers : ushort) : OpReturnCode +UpdateAll() : OpReturnCode +EnableCorrections() : OpReturnCode +DisableCorrections() : OpReturnCode +GetLastUpdateTimestamp(out update_timestamp : TimeRepresentation) : OpReturnCode +GetUpdateTimestampUncertainty(out update_timestamp_uncertainty : Uncertainty) : OpReturnCode

Figure 4-14 Transducer Block UML Class Diagram

The operations that are shown for this class are described in the following discussion.

- The *GetCorrectionMode* operation will be used to obtain the current state of the state machine for this object.
- The *GetNumberOfTransducerChannels* operation will be used to access the Meta-TEDS. This access will return the number of transducers that are implemented in the TIM that is physically connected to the NCAP.
- The *GetMinimumSamplingPeriod* operation will be used to access the Meta-TEDS. This access will return the time in seconds of the minimum sampling rate of the TIM as a whole, e.g. The minimum sampling rate for a global trigger.
- The *GetChannelParameterObjectChannelNumbers* will return the physical interface channel numbers for the implemented physical interface channels that correspond to the public transducer.
- The *GetUnrepresentedChannelNumber* operation will return an array of numbers with each array representing a channel present at the physical interface that is not represented by a public transducer in the NCAP.

- The *UpdateAll* operation will cause a global trigger to be applied to the transducer system. The operations that we have defined will be used to setup the transducer system and to apply top-level commands to the TIM as a whole.
- The *EnableCorrections* operation will be used for transitions within the state machine for this block. This transition will be from the uncorrected to the corrected state.
- The *DisableCorrections* operation will be used for transitions within the state machine for this block. This transition will be from the corrected to the uncorrected state.
- The *GetLastUpdateTimestamp* and *GetUpdateTimestampUncertainty* operations will not be implemented as they are optional according to the standard.

The behavior of this block is controlled by the basic state machine that was defined for all Block Objects. However, the Transducer Block sub-states this state machine so that it can apply corrections to the transducer data. This sub-stated state machine is shown in Figure 4-15.

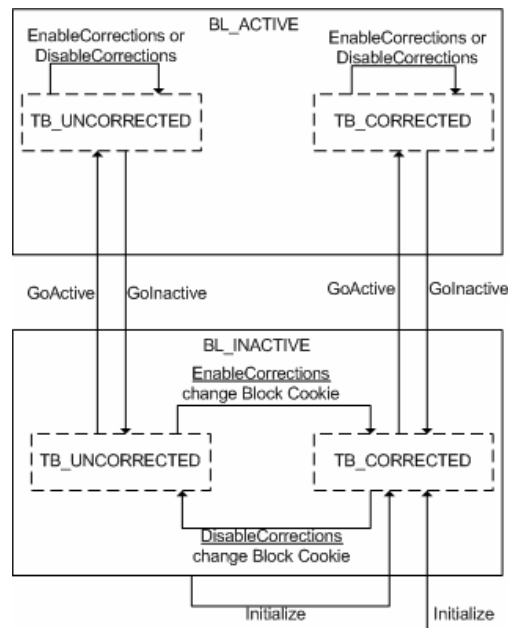


Figure 4-15 Correction state machine for a Transducer Block

Next, we give a brief overview about how this object models each of the transducer channels. In order to do this, the Transducer Block class exposes the channels as instances of a subclass of the Component class associated with the channel (e.g. Temperature sensor that is interfaced with an ADC is represented as a scalar parameter). This model is shown in Figure 4-16.

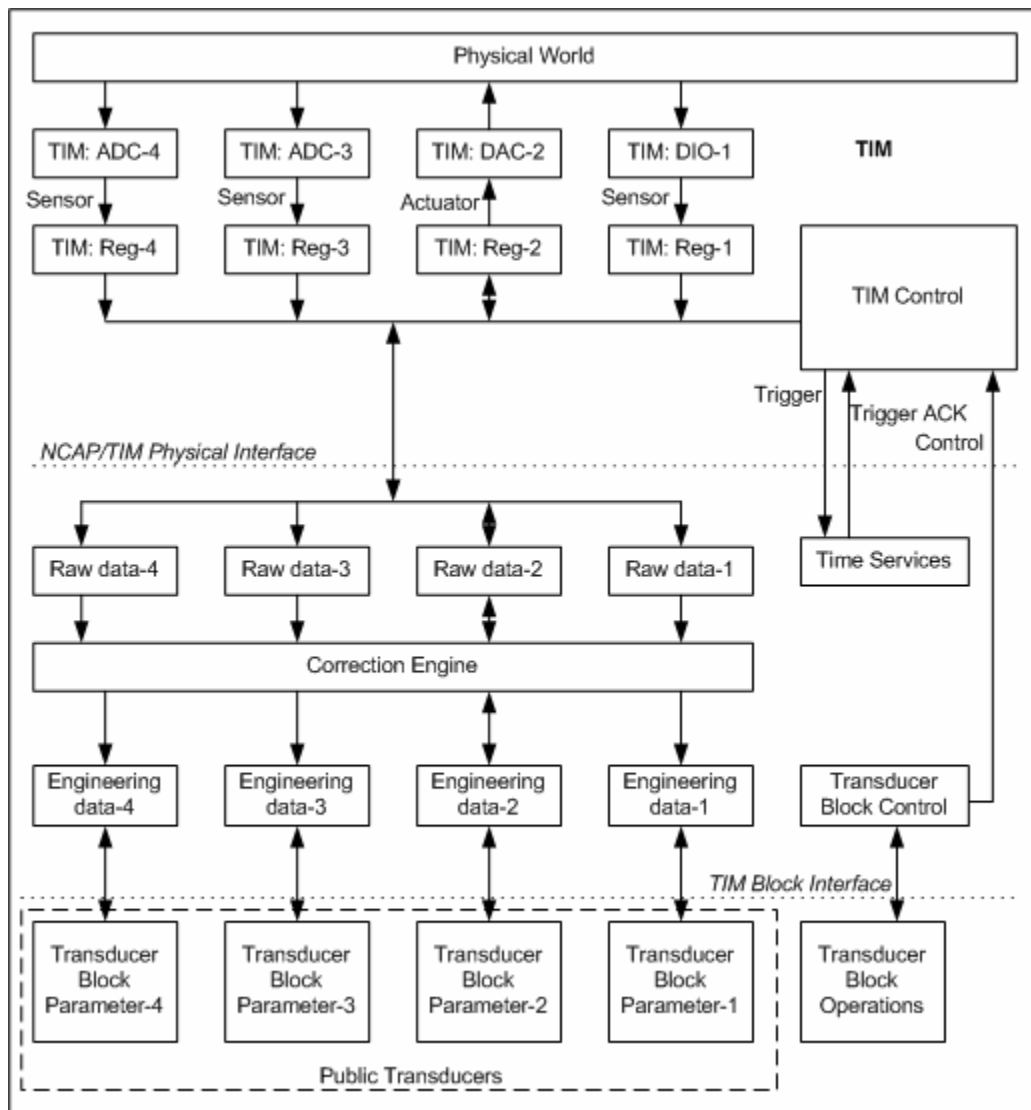


Figure 4-16 Model of a TIM and Transducer Block

The Transducer Block is modeled to contain a register, (raw data-1 to raw data-4 in the figure), that corresponds to each TIM register. These two sets of registers agree after the TEDS is

read and the NCAP knows the number/type of transducers in the TIM. A second set of registers, the engineering data registers of the figure, contains values mapping the contents of the raw data registers using the correction information that is provided by the TEDS. If the TIM does not support correction, then these two sets of registers may be modeled as a single set of registers. Next, we will discuss the Parameter class since it is used to model the individual transducers.

4.1.2.7 Parameter Classes

In order to apply individual triggering and access the Meta-TEDS we will use Parameter classes to model the transducers. So, there will be a Parameter class definition for each implemented transducer. This is important because individual triggering and accessing the Channel-TEDS is done by children of the Parameter class.

We begin by showing how individual triggering is generated by the NCAP. This is done through the Parameter With Update class. The structure for this class is shown in Figure 4-17 .

IEEE1451_ParameterWithUpdate
+_class_ID : ClassID +_description : String +_Class_name : String
+UpdateAndRead(out data : ArgumentArray) : OpReturnCode +ReadBlockUntilUpdate(out data : ArgumentArray) : OpReturnCode +WriteAndUpdate(in data : ArgumentArray) : OpReturnCode +WriteBlockUntilUpdate(in data : ArgumentArray) : OpReturnCode +GetLastTimeStamp(out last_timestamp : TimeRepresentation) : OpReturnCode #RegisterNotifyOnUpdate(in notification_operation, in registration_id : ushort) : OpReturnCode #DeregisterNotifyOnUpdate(in notification_operation, in registration_id : ushort) : OpReturnCode

Figure 4-17 Parameter with Update Class UML Diagram

The *UpdateAndRead* operation will be used to trigger a sensor channel.

The *WriteAndUpdate* operation will be used to trigger an actuator channel.

The *ReadBlockUntilUpdate*, *WriteBlockUntilUpdate*, *GetTimeStamp*, *RegisterNotifyOnUpdate*, and *DeregisterNotifyOnUpdate* operations are optional and will not be implemented in this design.

The behavior for the *UpdateAndRead* and *WriteAndUpdate* operations is shown in Figure 4-18. It is important to note, that the *UpdateAndRead* operation has no effect if it is applied to an actuator, and the *WriteAndUpdate* operation has no effect if it is applied to a sensor channel.

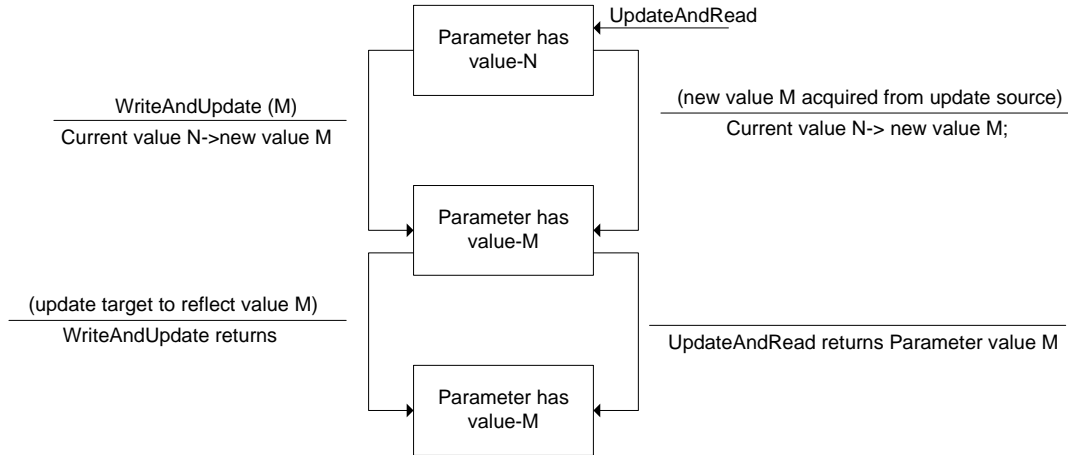


Figure 4-18 Time sequence behavior of UpdateAndRead and WriteAndUpdate

Next, we define the object class that will be used to access the Channel-TEDS. This class is the Physical Parameter class and its structure is shown in Figure 4-19.

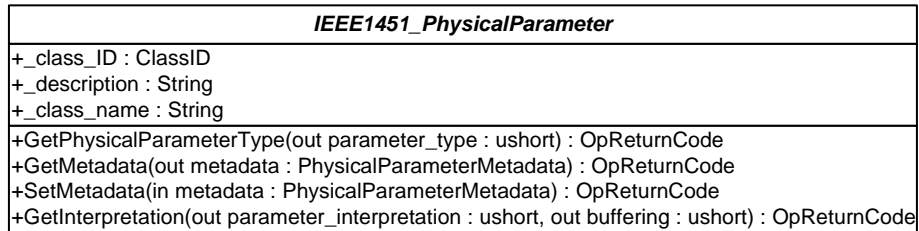


Figure 4-19 Physical Parameter UML Class diagram

The operations that are shown in the figure are further explained in the following discussion.

The *GetPhysicalParameterType* operation will be used to determine the type of transducer (sensor or actuator) that is being accessed.

The *GetMetadata* operation will be used to access the Channel-TEDS of the transducer channel that is being accessed.

The *SetMetaData* operation is defined as optional in the standard and will not be implemented.

The *GetInterpretation* operation will be used to obtain information about the transducer's data. For example, if it is an actuator value or sensor reading.

In this section we have defined a complete set of specifications that will be used to design the NCAP. This has been done by defining the different objects and formats defined by the information model of the 1451.1. The next section derives a set of specifications for the TIM that will be designed in compliance with the IEEE 1451 family of standards.

4.2 TRANSDUCER INTERFACE MODULE

The TIM is required to have the functionality of a smart transducer as is stated by the IEEE 1451 family of standards. The key element that will denote the TIM as smart is the TEDS since it will provide self-identification capabilities. Other required functionality for the TIM includes triggering behavior, interrupt generation, and status bits.

To meet the previously stated top-level requirements, the structure shown in Figure 4-20 will be used to design the TIM and its functionality. Note that the control unit will be responsible for NCAP communication and controlling the behavior of the TIM's objects. This module, as well as the other blocks can be implemented using a combination of hardware and software since the Excalibur chip provides these capabilities (on-chip FPGA and ARM processor). This configurability that the Excalibur chip provides will be exploited to enhance the performance of

the TIM's objects by designing them using hardware or software. This will depend on the advantages and disadvantages of each module implementation.

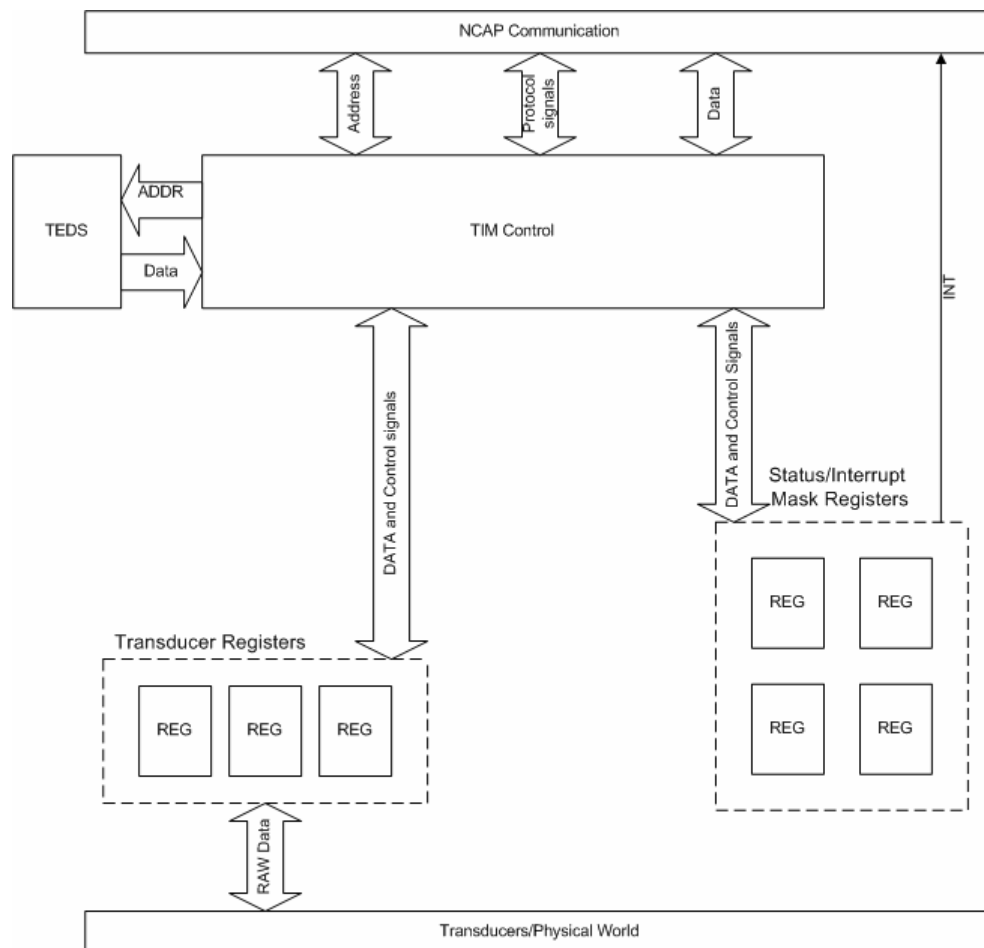


Figure 4-20 TIM Overall Structure

One of the key aspects of this architecture is that it will allow for a variable number of transducers. Therefore, designers can reuse this TIM with minimal effort, and configure it for their particular application. It is important to note that the number of transducers that can be interfaced to this TIM will be limited by the resources of the Excalibur chip. This chip comes in various sizes with the area ranging from 4,160 Logic Elements (LEs) in the smallest version (EPXA1), to 38,400 LEs in the largest chip (EPXA10). Similarly, the I/O ranges from 186 in the EPXA1 to 711 in the EPXA10.

4.2.1 Transducer Electronic Data Sheet

To comply with the requirements stated in Section 3.3.1 two mandatory TEDS blocks will be implemented. These blocks are the Meta-TEDS for the TIM entity and the Channel-TEDS for each individual implemented channel.

The TEDS structure will be based on the TEDS that is defined for the STIM of the 1451.2 standard. This is because at the top-level the TIM and STIM are similar since only one entity can make the connection with the NCAP. The following discussion goes into detail about the mandatory TEDS blocks (Meta and Channel) along with an explanation of each of the fields.

Table 4-3 shows the information that will be represented by the Meta-TEDS along with the definition of each field.

Table 4-3 Meta-TEDS Structure

Field No.	Description	Type	No. of Bytes
1	Meta-TEDS Length	U32	4
2	IEEE 1451 Standards Family Working Group Number	U8	1
3	TEDS Version Number	U8	1
4	Number of Implemented Channels	U8	1
5	Worst-Case Channel Data Model Length	U8	1
6	Worst-Case Channel Update Time (t_{wu})	F32	4
7	Worst-Case Channel Sampling Period (t_{wsp})	F32	4
8	Channel Groupings Data Sub-block Length	U16	2
9	Number of Channel Groupings = G	U8	1
10	Group Type	U8	1
11	Number of Group Members = N	U8	1
12	Member Channel Numbers List = M(N)	Array of U8E	N
13	Checksum for Meta-TEDS	U16	2

The explanation of the fields shown in the figure can be further explained as follows:

- The *Meta-TEDS Length* field specifies the total number of bytes in the TEDS.

- The *IEEE 1451 Standards Family Working Number* is used to denote if the transducer module belongs to the 1451.2, 1451.3 family among others. Since this implementation is different than the STIM and TBIM, this field shall be set to 255 which does not correspond to any approved standard and is reserved for future use.
- The *TEDS version number* specifies the version number of the TEDS. Again, for this implementation this field will be set to 255.
- The *Number of Implemented Channels field* specifies the number of channels implemented in the TIM. There can be up to 255 transducers in the TIM, so this field will be set to a number between 1 and 255.
- The *Worst-Case Channel Data Model Length* field specifies the maximum value of the Channel Data Model Length for all implemented channels. So if there are two transducers interfaced to an 8-bit and 12-bit ADC respectively, this field will be set to 12.
- The *Worst-Case Channel Update Time* field specifies the maximum value of the Channel Update Time for all implemented channels in seconds.
- The *Worst-Case Channel Sampling Period* specifies the maximum value in seconds, of the channel sampling period for all implemented channels.
- The *Channel Groupings Data Sub-Block Length* specifies the total number of bytes in the Channel Grouping data sub-block, which are the fields that follow it. If this value is zero, then no channel groupings are defined and there are no data bytes in the subsequent fields of the channel groups' data sub-block.

- The *Number of Channel Groupings* field specifies the number of discrete channel groupings defined in this TIM's Meta-TEDS.
- The *Group Type* field specifies the relationship between the channels comprising the specific group.
- The *Number of Group Members field* specifies the number of channels comprising the specific group.
- The *Member Channel Numbers List* specifies a one-dimensional array of 1 byte elements that represent the channel address for a member channel in the specific group.
- The *Checksum* field is the one's complement of the sum (module 2^{16}) of all the data structure's preceding bytes, including the initial length field and excluding the checksum field.

Similarly, Table 4-4 shows the structure for the Channel-TEDS.

Table 4-4 Channel-TEDS Structure

Field No.	Description	Type	No. of Bytes
1	Channel TEDS Length	U32	4
2	Calibration Key	U8	1
3	Channel Type Key	U8	1
4	Physical Units	UNITS	10
5	Lower Range Limit	F32	4
6	Upper Range Limit	F32	4
7	Worst-Case Uncertainty	F32	4
8	Channel Data Model	U8	1
9	Channel Data Model Length	U8	1
10	Channel Model Significant Bits	U16	2
11	Channel Update Time (t_u)	F32	4
12	Channel Sampling Period (t_{sp})	F32	4
13	Checksum for Channel TEDS	U16	2

The explanation for the fields shown in the figure is as follows:

- The *Channel TEDS length* field specifies the total number of bytes in the Channel TEDS data block excluding this field.
- The *Calibration Key* field specifies the calibration capabilities of the TIM. This field specifies if calibration will be done within the TIM block or if it needs to be done in the NCAP.
- The *Channel Type Key* field specifies the channel transducer type (sensor or actuator).
- The *Physical Units* field specifies the physical units that apply to the transducer data of the particular channel. This field applies to transducer data after correction for sensors, or before correction for actuators.
- The *Lower Range Limit* field has different meanings for sensors and actuators. For sensors, it specifies the lowest valid value for transducer data after correction is applied, so if the corrected transducer data lies below this limit, it may not comply with TIM specifications set by the manufacturer. For actuators, this field specifies the lower valid value for transducer data before correction is applied.
- The *Upper Range Limit* field specifies the maximum valid value for a sensor's data after correction is applied, or the maximum valid value for an actuator's data before correction is applied.
- The *Worst-Case Uncertainty* field specifies the “Combined Standard Uncertainty”⁽²²⁾.
- The *Channel Data Model* field describes the data model used when reading or writing data to the transducer. This field specifies if the model is an integer,

single-precision floating point number, double-precision floating point number, or a fraction.

- The *Channel Data Model Length* field specifies the number of bytes in the representation of the selected channel data model.
- The *Channel Model Significant Bits* field specifies the numbers of bytes that are significant. For example if data from a transducer comes from a 12-bit ADC, then this field will be set to 2.
- The *Channel Update Time* field specifies the maximum time in seconds, between the receipt of a trigger and a trigger acknowledge for this channel.
- The *Channel Sampling Period* field specifies the minimum sampling period of the channel transducer unencumbered by read or write considerations. Typically, this time is limited by the ADC/DAC conversion times.

4.2.2 NCAP Communication

Because only one TIM can make the connection to a single NCAP as was stated in the requirements Section 3.2.2, this top-level connection will resemble the STIM/NCAP communication of the 1451.2 standard. Therefore, the command structure will have a functional and channel address associated with it. Each individual implemented transducer will have a channel address associated with it. So, the channel address will be large enough such that it can accommodate the maximum number of transducers (255) that can be implemented in the TIM. The functional address will be used to denote the command that is being sent by the NCAP to the TIM. The length of this field will be eight bits, which will give the designer enough room for the

mandatory set of commands, as well as leaving some room for future expansions. Table 4-5 summarizes the list of commands that will be implemented.

It is important to note that the trigger operation is implemented as a command, which is different than what is done in the STIM. In the STIM a single line in the TII (10-wire serial connection) is dedicated for the trigger operation, so to trigger a sensor the user has to first set the channel address in a separate instruction and then issue the trigger. For the case of actuators, the user has to write the actuator data, write the channel address, and then the user can apply the trigger command. Since we are eliminating this serial connection between NCAP/TIM there is no need for this dedicated line. Thus improving the speed performance since users can issue a trigger and channel address in a single command.

Table 4-5 TIM's Commands

Functional Address	CHANNEL_ZERO command	Single Channel Command
0	Read Meta-TEDS	Read Channel TEDS
1	Write CHANNEL_ZERO interrupt mask	Write channel standard interrupt mask
2	Read CHANNEL_ZERO interrupt mask	Read channel standard interrupt mask
3	Read CHANNEL_ZERO status	Read channel status
4	CHANNEL_ZERO Trigger	Transducer Channel trigger
5	Write Transducer Channel Data	Write Transducer Channel Data
6	Read Transducer Channel Data	Read Transducer Channel Data
7	Reset	Reserved
MSB	For future use	For future use

The following is a detailed explanation about the commands and their top-level functionality.

- The *Read Meta-TEDS* and *Read Channel-TEDS* commands return the complete structure of the TEDS blocks to the NCAP.
- The *write interrupt mask* commands allow the NCAP to set the interrupt mask of the implemented transducer channels in the TIM.
- The *Read Status operation* returns the status of the particular channel that is addressed.
- The *trigger* command is used to sample/set the transducer channel that is addressed. Note that if CHANNEL_ZERO is used as the address, then every implemented channel in the TIM is triggered.
- The *write transducer channel data* command is used to send the data to the actuator's register that will be used when the channel is triggered.
- The *read transducer channel data* command is used to access the sensor information from the last sampled event.
- The *reset* command is used to put all the TIM's objects in their default power-on state.

4.2.3 Trigger

The TIM is required to handle both individual and global triggering. When triggered, a sensor will be sampled and an actuator will acquire a new data set. The functionality for triggering will be handled by the control unit that was shown in Figure 4-20. Therefore, this unit will be responsible for setting the control signals of the transducer registers, as well as the data

converters that are associated it with the transducers. The control unit will go through the following sequence of events when a trigger command is issued.

First, the control unit will decode the information and send it to the particular channel so that the trigger can be executed. Then the acknowledge signal will be generated as follows. Note that it is generated differently for sensors and actuators.

For sensors, the trigger ACK signal will be generated when the ADC finishes conversion and the TIM latches this data onto its particular data register. If the sensor is connected to digital IO then the ACK signal will be generated when the data is latched onto its data register.

For actuators, the trigger ACK will be sent upon receipt of a trigger function when interfaced with digital IO. If the actuator is interfaced to a DAC then the ACK signal will be generated after the device finished conversion.

During global triggers, the CHANNEL_ZERO ACK will be generated when every channel has been successfully sampled/set as defined by the particular transducer. Also, each individual channel will generate an ACK when it has successfully completed the trigger operation.

4.2.4 Status and Interrupts

There are four mandatory status bits for CHANNEL_ZERO, and three status bits for individual channels that will be implemented. These bits can be summarized as follows. For CHANNEL_ZERO the implemented status bits will be Trigger ACK, Invalid Command, TIM operational and Corrections enabled/disabled. For individual channels, the implemented status bits will be Trigger ACK, Channel Operational, and Corrections enabled/disabled.

These status bits along with their corresponding interrupt mask can be implemented using registers giving the NCAP an easy way to access this information. This is because registers are assigned a memory location in the embedded stripe of the Excalibur chip and are accessible by reading/writing to the pre-assigned memory location.

These registers can be 32 bits giving the user a wide range of status bits that were not defined in the requirements but may be added for more complex systems. There will be a register for each implemented channel in the TIM.

The interrupts will be generated by a combination of status and interrupt mask registers similar to how is done in the 1451.2 STIM³. This scheme is slightly modified with the difference being that in the standard the LSB of the status register is reserved for the service request bit. This bit is a logical OR of all the AND operations in the status/interrupt mask combination, so when this bit is set an interrupt signal is generated. The reason why this is done is because an individual line in the TII protocol is reserved for the interrupt signal much like what is done for the trigger. Again, because we are eliminating this serial protocol there is no need for this single line. Instead, each implemented channel may have up to 32 different interrupt signals that can be sent to an interrupt controller in order to process the requests. Figure 4-21 shows the interrupt generation scheme that will be used.

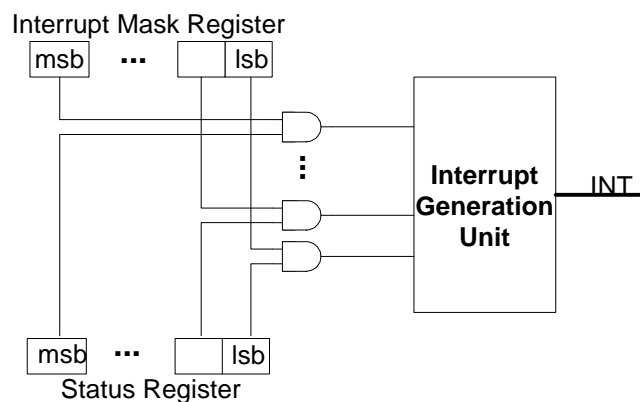


Figure 4-21 Interrupt Masking

The Interrupt Generation Unit shown in the figure is a priority encoder that will take the result of the AND operations and encode them into the bit size of the interrupt controller that will be used.

4.3 SUMMARY AND TESTS

The functionality that is stated in this section can be summarized in Table 4-6. Note that the table shows inputs and outputs to the application system.

Table 4-6 System Summary

Name	System on a Chip Solution for Smart Networked Transducers
Purpose	Single chip implementation of the IEEE 1451 family of standards
Inputs	Data read from sensor(s), Network information sent to the NCAP
Outputs	Data set sent to actuator(s), System results sent to network(s)
Functions	<p>Initialization: Initialize entire control system (includes both NCAP and TIM). NCAP reads TEDS and initializes its control logic. While the TIM will be initialized upon power-up by local intelligence within the block.</p> <p>NCAP<->TIM communication: The NCAP sends a command to the TIM and it responds accordingly.</p> <p>Trigger: NCAP sends a trigger command to the TIM which decodes it and executes it for either a sensor or actuator, or both (global triggering).</p> <p>Interrupts: Interrupts generated by TIM that are serviced by NCAP</p> <p>NW communication: Communication between NCAP and network using the publish/subscribe or client-server method.</p>

The functionality shown in the table needs to be tested in order to ensure compliance with the standard. In order to test the system, a proof of concept application with two sensors

(temperature and light) and LEDs to simulate an actuator will be built. Note that the transducers will be designed to meet the electrical and timing specifications of the Excalibur chip.

This test application will consist of a closed-loop control system controlled by the NCAP. So, the main program will execute in a loop where the functionality of the system will be tested. In doing this, the NCAP will test the entire command structure of the TIM as well as the network communications by the use of the objects defined for an IEEE 1451.1 NCAP. The setup for this control application will be as follows:

Initialization: During this stage the network port information will be mapped to pre-defined objects that are set at compile-time. Then, we will apply a *reset* command to the TIM in order to test that required operation. Next, the TEDS will be read and the different transducer objects (on the NCAP-side) will be instantiated from this information. This interaction will prove that the TEDS structure can be successfully read and that the different transducer objects in the NCAP's object model can be instantiated correctly.

After verifying that there were no errors in the TEDS access and object instantiation, the NCAP can be initialized. This initialization will consist of calling the *Initialize* and *GoActive* operations and making sure that they are executed properly. The interrupt masks of the TIM will also be set during this phase using application-specifics commands. This will be done to test the TIM command *Write Interrupt Mask* for CHANNEL_ZERO and individual channels. Also, any initialization that needs to be done for the network hardware or other non-standard blocks will be done in this stage.

Main Program: The next step is to test the application by showing interactions between the network, NCAP, and TIM. To do this, a simple application will be configured that constantly

triggers the transducers by using the *UpdateAndRead*, *WriteAndUpdate*, and *UpdateAll* operations previously shown in Sections 4.1.2.6 and 4.1.2.7

This completely tests the mandatory command set of the TIM, since every time that a trigger command is issued the TIM will generate an interrupt (Trigger ACK). Then, the NCAP will service the interrupt, which will test the *Read Status Register* and *Read Transducer Channel Data* TIM operations.

Next, a dummy packet will be encoded into an *ArgumentArray* for network communications using both models. To test both communication models, two different dummy packets will be used for client-server and publish-subscribe communications.

This loop will test the entire command set of the TIM, the top-level functionality of the NCAP, and that a network can be plugged in according to the standard. Therefore, we will show that the high-level requirements of the IEEE 1451 solution are met.

5.0 DESIGN

The system design is done using a combination of hardware (on-chip and off-chip peripherals) and software (C for the ARM922T) blocks. These blocks are used to design a system that meet the requirements stated in Chapter 3.0 using the specifications previously stated in Chapter 4.0.

We begin the design by establishing the NCAP/TIM communication in such a way that it meets the performance requirement, which states that this connection shall be faster than the TII of the 1451.2 STIM. To meet this requirement, the NCAP/TIM connection is established using the AMBA AHB that is provided by the Excalibur chip. This connection can reach speeds up to 160 MHz. Therefore, using the AHB the sampling rate for a system with 10 transducers (5 sensors and 5 actuators) would be 374 KHz (assuming an ADC conversion rate of 2.5 μ Sec), as opposed to the 51 Hz that the 1451.2 connection yields under the same configuration. To use this bus communication, the TIM's top-level functionality is designed as a peripheral on the bus using custom logic (PLD). The rest of the modules for this IEEE 1451 solution will be discussed in the following sub-sections of this chapter.

5.1 NETWORK CAPABLE APPLICATION PROCESSOR

In this section we will focus on the software blocks that are specified by the standard's information model. To do this, we first give an overview of an IEEE 1451.1 NCAP shown in Figure 5-1.

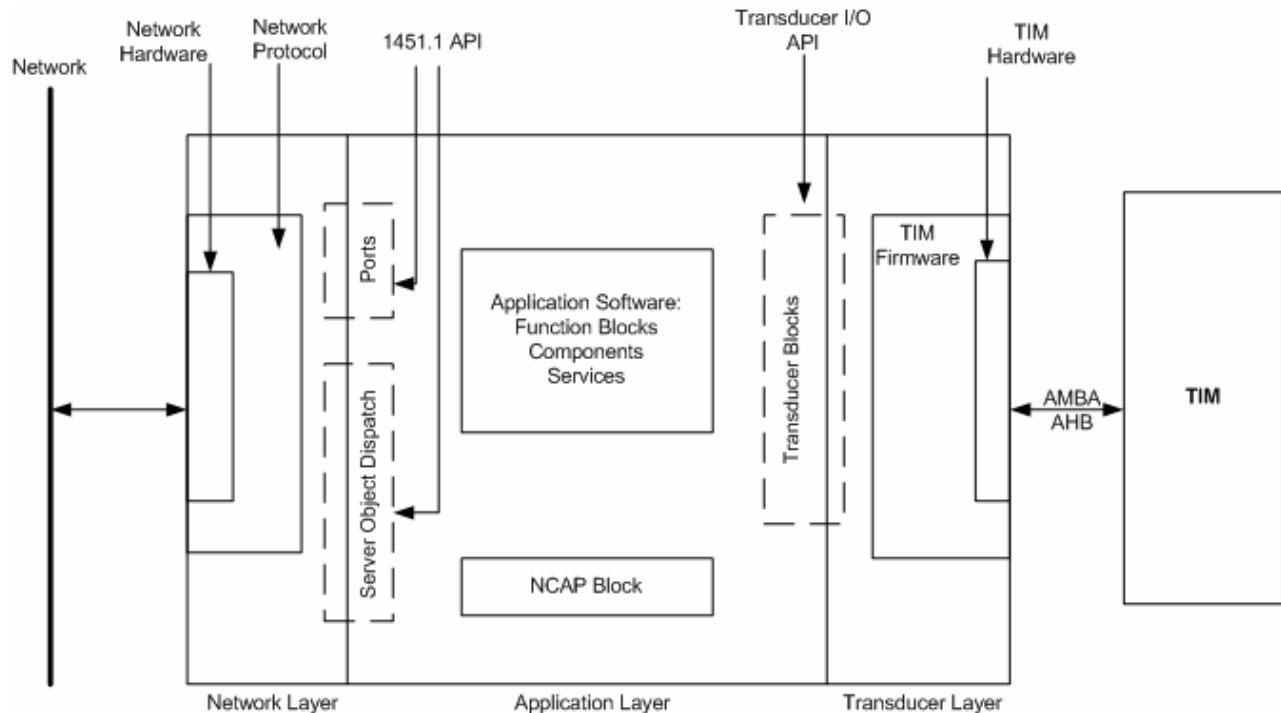


Figure 5-1 NCAP Overview

Note that in the application layer, we have an NCAP Block that is used to control the experiment, and we also have the application-specific software (given by Function Blocks). Both of these blocks are inherited from the Block class. Then, in the network and transducer layers, we have APIs. These APIs are the hooks that are provided to communicate with the network and TIM that are interfaced with the NCAP. The 1451.1 API provides the necessary hooks for network communication. These hooks are specifically provided by the network ports and the Server Object Dispatch given by the Entity, Client Port, Publisher Port, and Subscriber Port

object classes. Note that the Entity and Client Port classes are used for client/server communications; while the Publisher Port and Subscriber Port classes are used for publish/subscribe communications. On the other hand, the Transducer I/O API provides the hooks for TIM communication. These hooks are given by the Transducer Block class (for the TIM as a whole) and by Parameter classes (for individual transducers within the TIM).

Next, we discuss the programming language that will be used for this NCAP design. The software blocks are designed in C, which is not an object-oriented programming language. This presents some issues when designing the class hierarchy and object owning relationships previously shown in Figure 4-1. To solve this problem the objects are designed as structures and the class hierarchy is designed by defining a super class that acts as the parent. This is shown in Figure 5-2.

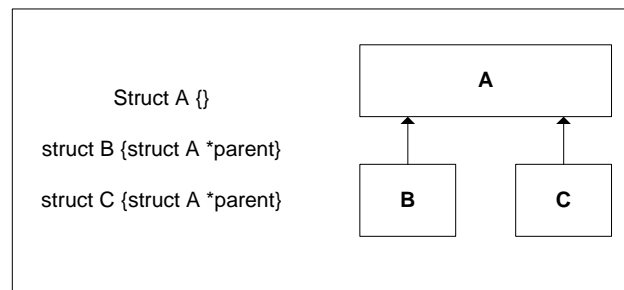


Figure 5-2 Class Hierarchy Implementation

In the figure, we design a super class A with two child nodes B and C. Using pointers the child node “points” to its parent, which allows us to implement the parent/child relationship that is described as the class hierarchy. In order to use this relationship, we use the scheme shown in Figure 5-3 to instantiate the structures using C.

```
Struct A super;  
Struct B child_one;  
Struct C child_two;  
  
child_one.parent = *super;  
child_two.parent = *super;
```

Figure 5-3 Structure instantiation in C

Another issue that needs to be resolved is that each object class is supposed to have operations that only the particular object and its children can call. To solve this problem, we include the structure node as a parameter of the operation, e.g. void foo (struct *A, int B). Note that in the example an instance of structure A needs to be provided to call function *foo*. Using the two schemes previously discussed, we can design the NCAP using its object orientation with the C programming language.

Next, we will design the complete design of the information model that was previously described in the specifications chapter. It is important to note, that as was mentioned before by implementing the different objects and formats, the standard is met and users can use this implementation for their particular application.

5.1.1 Datatype Mapping

The datatypes that are defined in the standard and that were defined in the specifications chapter need to be mapped to the C programming language. In order to do this, we use some pre-defined C types as well as create new types that are used in the design of this IEEE 1451.1 NCAP. Table 5-1 summarizes this datatype mapping.

Note that the colon operator is used to denote an 8-bit integer. This is done because C for the ARM does not provide an 8-bit integer type and using the colon operator denotes the type as only 8-bits of length.

Table 5-1 Datatype to C Mapping

Datatype	C Mapping
Boolean	Defined a new structure that defined a TRUE as a 1 and FALSE as a 0
Integer8	Short : 8
UInteger8	Unsigned short : 8
Integer16	Short
UInteger16	Unsigned short
Integer32	Int
UInteger32	Unsigned int
Integer64	Long
UInteger64	Unsigned long
Float32	Float
Float64	Double
Octet	Char

For all the types that are defined in the standard, we create a header file that contains all the structure information of the different objects in the NCAP. It is important to note that arrays are implemented as pointers. This way we can use the *malloc()* and/or *talloc()* operation to dynamically allocate the memory that is needed for the array representation. Next, we show the design of the top-level classes for this NCAP design.

5.1.2 IEEE 1451.1 API

This API is used to hide the communication details of network communications. Therefore, this API provides the different hooks that can be used regardless of the network that is employed. Next, we will design the objects that are responsible for network communications using the client/server and publish/subscribe models.

5.1.2.1 Client-Server Network Communications

In this section, we will go into detail about the design of the operations of the Entity and Client Port classes that were previously shown in the specifications chapter. We begin with the Entity class whose structure is shown in Figure 5-4.

```
typedef struct {  
    IEEE1451_Root *super;  
    ClassID class_ID;  
    IEEE1451_String description;  
    IEEE1451_String parent_name;  
    ObjectTag object_tag;  
    ObjectID object_ID;  
    char *object_dispatch_address;  
}IEEE1451_Entity;
```

Figure 5-4 Entity Class Structure in C

The *super* field is a pointer to the Entity class, which is used to denote the parent/child relationship of the class hierarchy. The *class_ID*, *description*, and *parent_name* fields are part of the class header that was previously mentioned in the specifications chapter. Therefore, the *class_ID* encodes the position of this object in the class hierarchy. The *description* is the class name of this object. The *parent_name* is the class name of the parent class (in this case the entity class). Note that for the *description* and *parent_name* parameters the standard provides an enumeration that is used.

The *object_tag* field contains the *ObjectTag* (logical endpoint for server-side communications) of this particular object. Note that this value is unique for each object in an NCAP application system.

The *object_ID* field contains the *ObjectID* (used to unambiguously distinguish the object from any other object) of this particular object.

The *object_dispatch_address* is used to represent the network-specific address of the underlying network.

The following discussion goes into detail about the design of the operations for this class along with an explanation of how they are designed.

(1) *OpReturnCode GetObjectTag(IEEE1451_Entity *a,*
/ out */ ObjectTag *object_tag)*

This operation returns the *object_tag* parameter of the Entity class.

(2) *OpReturnCode SetObjectTag(IEEE1451_Entity *a,*
/ in */ ObjectTag object_tag)*

This operation initializes/modifies the *object_tag* parameter of the Entity class

(3) *OpReturnCode GetObjectID(IEEE1451_Entity *a,*
/ out */ ObjectID *object_id)*

This operation returns the *object_ID* parameter of the Entity class.

(4) *OpReturnCode GetObjectName(IEEE1451_Entity *a,*
/ out */ IEEE1451_String *object_name)*

This operation returns the *description* parameter of the Entity class.

(5) *OpReturnCode GetDispatchAddress(IEEE1451_Entity *a,*
/ out */ ObjectDispatchAddress *dispatch_address)*

This operation returns the *object_dispatch_address* parameter of the Entity class.

(6) *OpReturnCode GetOwningBlockObjectTag(IEEE1451_Entity *a,*
/ out */ ObjectTag *owning_block_object_tag)*

This operation returns the *ObjectTag* of the owning block object following the relationship that was described in the specifications chapter.

(7) *OpReturnCode GetObjectProperties(IEEE1451_Entity *a,*
/ out */ ObjectProperties *object_properties)*

This operation returns the *object_properties* parameter of the Entity class.

(8) *ClientServerReturnCode Perform(IEEE1451_Entity *a,*
/ in */ unsigned short server_operation_id,*
/ in */ ArgumentArray server_input_arguments,*
/ out */ ArgumentArray server_output_arguments)*

This operation is the server-side construct for client-server operations. The pseudo-code for this operation is shown in Figure 5-5. Note that the server operation that is being targeted by the client is provided by the *server_operation_id* that is generated by the network infrastructure after it de-marshals the network packet. For the *server_operation_id* the standard defines an enumeration, so that this value can be matched to every operation within an NCAP.

Also, when the operation times out, we return an *ArgumentArray* of size zero which is done to follow recommendations that are given by the standard.

Next, we discuss the design of the client-side for these types of network communications. This is done through the Client Port class. The design of this class begins with the structure that is created for this object. This structure is shown in Figure 5-6.

```

Decode the server_input_arguments

If execution mode is return value then
    Call the operation that is meant for the server
    Wait for server operation to complete

    If timeout then
        Return operation Timed out

    Else then
        Encode outputs into arguments
        Send the new packet to the network infrastructure
        Return successful operation
Else then
    Call the operation that is meant for the server
    Return successful operation

```

Figure 5-5 Perform() Operation pseudo-code

```

typedef struct {
    IEEE1451_BaseClientPort *super;
    ClassID class_ID;
    IEEE1451_String description;
    IEEE1451_String parent_name;
    ObjectTag object_tag;
    ObjectID object_ID;
    unsigned short *block_state;
} IEEE1451_ClientPort;

```

Figure 5-6 Client Port Class Structure in C

The only attribute that is specific to this class is the *block_state* parameter, which is used to “point” to the NCAP Block’s state machine.

Next, we will design the operations that are defined for this class.

```

(1) ClientServerReturnCode Execute(IEEE1451_ClientPort *a,
    /* in */ unsigned short execute_mode,
    /* in */ unsigned short server_operation_id,
    /* in */ ArgumentArray server_input_arguments,

```

```
/* out */ArgumentArray server_output_arguments)
```

This operation provides the client-side functionality for client-server communications. The pseudo-code for this operation is shown in Figure 5-7.

```
If execution_mode is return then  
    Send the packet to the network infrastructure  
    Wait for return from server  
  
    If timeout then  
        Return operation timed out  
    Else then  
        Return successful operation and server output info  
  
Else  
    Send the packet to the network infrastructure  
    Return successful operation
```

Figure 5-7 Execute() Pseudo-Code

It is important to note that the call to the network infrastructure is the “hook” to communicate with the underlying network. Therefore, in order to use this operation for a custom implementation/application, we have to change the *Send the packet to the network infrastructure* to call the particular network library that is used in the application. Also, from the specifications chapter we know that there are two types of execution modes for this operation (blocking and “send and forget”).

Therefore, we first check to see which mode we will use. After verifying this, we then send the packet to the network infrastructure (network-specific operation) so that it can be marshaled onto the on-the-wire format of the underlying network. Then, if the execution mode is “send and forget” (*execution_mode* parameter is equal to one) we are done and return to the application. In the case that the

execution mode is the blocking one (*execution_mode* parameter is equal to zero), then we wait until the server decodes the operation and returns the results from its operation. Also, in this case (execution mode is blocking), the operation timeouts if the server-side does not respond within a certain time that is set between the two NCAPs.

Note that the packet that is sent to the network infrastructure consists of the *ArgumentArray server_input_arguments*. Therefore, the designer needs to customize an operation in the network library that decodes this datatype and puts it on the format of the underlying network.

5.1.2.2 Publish-Subscribe Network Communications

The publish-subscribe network communications are executed using the Publisher Port class and the Subscriber Port class. We begin by showing the generated structure of the Publisher Port class. This structure is shown in Figure 5-8.

```
typedef struct {  
    IEEE1451_BasePublisherPort *super;  
    ClassID class_ID;  
    IEEE1451_String description;  
    IEEE1451_String parent_name;  
    ObjectTag object_tag;  
    ObjectID object_ID;  
    unsigned short *block_state;  
}IEEE1451_PublisherPort;
```

Figure 5-8 Publisher Port Class Structure in C

The parameters for this class have the same meaning as what has been defined for the client/server classes in the previous section.

Next we design the operations that are defined for this class along with an explanation of how they are designed.

```
(1) OpReturnCode Publish(IEEE1451_PublisherPort *a,  
/* in */ ArgumentArray publication_contents)
```

This operation provides a “hook” to communicate with the underlying network. It does this by sending the *publication_contents* to the network infrastructure (network-specific) so that it can be marshaled and sent across the underlying network. Therefore, to customize this operation we need only to call the network-specific operation (provided by the network infrastructure). For example, the C code for this operation can be written in a single line that calls the network-specific operation that marshals the *ArgumentArray* onto the on-the-wire format of the network that is employed.

Next, we discuss the Subscriber Port class since it provides the subscriber-side functionality for these types of network communications. The structure for this class is shown in Figure 5-9.

```
typedef struct {  
    IEEE1451_Service *super;  
    ClassID class_ID;  
    IEEE1451_String description;  
    IEEE1451_String parent_name;  
    ObjectID object_ID;  
    unsigned short *block_state;  
    SubscriptionQualifier subscription_qualifier;  
    unsigned short subscription_key;  
    PubSubDomain subscription_domain;  
} IEEE1451_SubscriberPort
```

Figure 5-9 Subscriber Port Class Structure in C

There are three parameters that are specific to this class. These types are further explained in the following discussion.

The *subscription_qualifier* field is used to determine which publications will be accepted.

The *subscription_key* field is used to configure the subscriptions to the publications.

The *subscription_domain* field is used to define a set of candidate publications to be accepted by the port.

Next, we will design the different operations that are defined for this class along with an explanation of how they are designed.

(1) *OpReturnCode SetSubscriptionQualifier(IEEE1451_SubscriberPort *a,*
/ in */ SubscriptionQualifier subscription_qualifier)*

This operation initializes/modifies the *subscription_qualifier* parameter of the Subscriber Port class.

(2) *OpReturnCode GetSubscriptionQualifier(IEEE1451_SubscriberPort *a,*
/ out */ SubscriptionQualifier *subscription_qualifier)*

This operation returns the *subscription_qualifier* parameter of the Subscriber Port class.

(3) *OpReturnCode GetSubscriptionKey(IEEE1451_SubscriberPort *a,*
/ out */ unsigned short *subscription_key)*

This operation initializes/modifies the *subscription_qualifier* parameter of the Subscriber Port class.

(4) *OpReturnCode SetSubscriptionDomain(IEEE1451_SubscriberPort *a,*
/ in */ PubSubDomain subscription_domain)*

This operation initializes/modifies the *subscription_domain* parameter of the Subscriber Port class.

(5) *OpReturnCode GetSubscriptionDomain(IEEE1451_SubscriberPort *a,*
/ out */ PubSubDomain subscription_domain)*

This operation returns the *subscription_domain* parameter of the Subscriber Port class.

5.1.3 Transducer I/O API

This API hides the communication details with the transducers that are physically connected to the NCAP. This is done through the Transducer Block Class (for the TIM as a whole), and by means of Parameter Classes (for the individual transducers). In this section, we will show the design of the different objects and operations that provide this level of abstraction.

5.1.3.1 Transducer Block Class

The design of this class begins with the structure that is created for this object. This structure is shown in Figure 5-10.

```
typedef struct {  
    IEEE1451_BaseTransducerBlock *super;  
    ClassID class_ID;  
    IEEE1451_String description;  
    IEEE1451_String parent_name;  
    ObjectTag object_tag;  
    ObjectID object_ID;  
    unsigned short *block_state;  
    unsigned short sub_state : 8;  
}IEEE1451_TransducerBlock;
```

Figure 5-10 Transducer Block Class Structure in C

Note that the different parameters that are designed for this class are similar to those of the other Block classes that we have discussed in this chapter. So, for an explanation on those parameters refer to previous sections. Next, we design the different operations that are defined for this class.

(1) *OpReturnCode EnableCorrections(IEEE1451_TransducerBlock *a)*

This operation is used to transition in the state machine of this class. This transition consists of going from the TB_UNCORRECTED to the TB_CORRECTED state. The pseudo-code for this operation is shown in Figure 5-11.

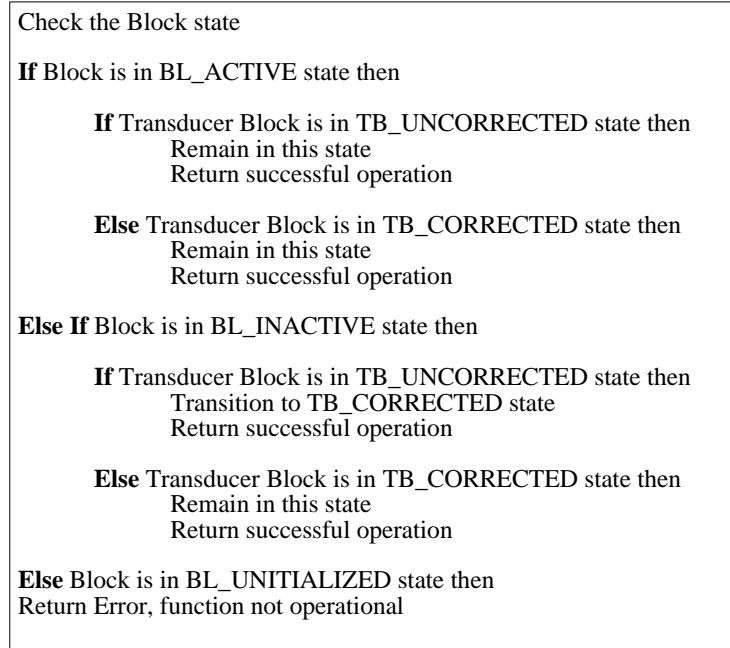


Figure 5-11 EnableCorrections() operation pseudo-code

(2) *OpReturnCode DisableCorrections(IEEE1451_TransducerBlock *a)*

This operation is used to transition in the state machine of this class. This transition consists of going from the TB_UNCORRECTED to the

TB_CORRECTED state. The pseudo-code for this operation is shown in Figure 5-12.

(3) *OpReturnCode GetCorrectionMode(IEEE1451_TransducerBlock *a,*
/ out */ unsigned short *correction_mode)*

This operation returns the *sub_state* field of this object class.

(4) *OpReturnCode GetNumberOfTransducerChannels(TransducerBlock *a,*
/ out */ unsigned short *number_of_transducer_channels)*

This operation returns the field of the Meta-TEDS that signals the number of implemented transducer channels in the TIM. Note that this TEDS information is obtained accessing a header file that is provided at compile-time. For example, in this operation we write a single line of code as is shown below:

**number_of_transducer_channels = CHANNEL_ZERO.channels;*

```

Check the Block state
If Block is in BL_ACTIVE state then
    If Transducer Block is in TB_UNCORRECTED state then
        Remain in this state
        Return successful operation
    Else Transducer Block is in TB_CORRECTED state then
        Remain in this state
        Return successful operation
Else If Block is in BL_INACTIVE state then
    If Transducer Block is in TB_UNCORRECTED state then
        Remain in this state
        Return successful operation
    Else Transducer Block is in TB_CORRECTED state then
        Transition to TB_UNCORRECTED state
        Return successful operation
Else Block is in BL_UNINITIALIZED state then
    Return Error, function not operational

```

Figure 5-12 DisableCorrections() operation pseudo-code

(5) *OpReturnCode GetMinimumSamplingPeriod(IEEE1451_TransducerBlock *a,*
/ out */ TimeRepresentation *minimum_sampling_period)*

This operation returns the time in seconds of the minimum sampling rate of the entire TIM. In order to design this operation, we write a single line of code that accesses the TEDS header file, this line is as follows:

**minimum_sampling_period = CHANNEL_ZERO.sampling_period;*

(6) *OpReturnCode GetChannelParameterObjectTags(TransducerBlock *a,*
/ in */ unsigned short channel_number,*

/ out */ ObjectTagArray *parameter_object_tags)*

This operation returns an array of *ObjectTags* that identify the public transducers that use the channel that is being accessed.

(7) *OpReturnCode GetParameterObjectChannelNumbers (TransducerBlock *a,*
/ in */ ObjectTag parameter_object_tag,*

/ out */ unsigned short *channel_numbers)*

This operation checks the *ObjectTag* of the class and returns the channel number of the particular channel that is being accessed.

(8) *OpReturnCode GetUnrepresentedChannelNumbers(TransducerBlock *a,*
/ out */ unsigned short *unrepresented_channel_numbers)*

This operation returns a zero because all channels that are physically connected to the NCAP are represented.

(9) *OpReturnCode UpdateAll(IEEE1451_TransducerBlock *a)*

This operation is used to apply a global trigger to every transducer channel that is connected to the NCAP. The pseudo-code for this operation is shown in Figure 5-13.

```
Trigger CHANNEL_ZERO  
Waiting for Trigger ACK  
  
If timeout then  
    Return Error Message  
Else then  
    Return successful operation
```

Figure 5-13 UpdateAll() Pseudo-code

Note that this operation sends a global trigger command through the AHB to the TIM. Therefore, because it is a global trigger, this operation does not need to be customized. Also, this operation times out if the worse-case channel update time of the Meta-TEDS is exceeded.

5.1.3.2 Parameter Classes

In this section we will discuss the classes that are used for individual channel triggering as well as Channel-TEDS access. We begin with the Parameter With Update class since this class is used to apply individual channel triggering for sensors and actuators. The structure that is generated for this class is shown in Figure 5-14.

In this section we will focus on the *UpdateAndRead*, *WriteAndUpdate* and *GetMetadata* operations. This is because those three operations are used to apply an individual trigger as well as access the Channel-TEDS.

```
typedef struct {
    IEEE1451_Parameter *super;
    ClassID class_ID;
    IEEE1451_String description;
    IEEE1451_String parent_name;
    ObjectTag object_tag;
    ObjectID object_ID;
    unsigned short *block_state;
}IEEE1451_ParameterWithUpdate;
```

Figure 5-14 Parameter With Update Class Structure in C

(1) *OpReturnCode UpdateAndRead(IEEE1451_ParameterWithUpdate *a,*

/ out */ArgumentArray data)*

This operation is used to apply an individual trigger to a sensor channel. The pseudo-code for this operation is shown in Figure 5-15.

```
Check to see what type of transducer we are dealing with
If sensor then
    Trigger the sensor
    Wait for Trigger ACK

    If timeout then
        Return error message
    Else then
        Get transducer data
        Encode data into an Argument Array
        Return successful operation

Else then
    Print "Tried to read an Actuator channel so nothing happens"
    Return successful operation
```

Figure 5-15 UpdateAndRead() pseudo-code

Note that the first thing that we do is to check and see what type of transducer it is. After verifying that it is a sensor, we then proceed to trigger the sensor channel. Then we wait for the trigger ACK signal and if it is not received by the

NCAP within the channel update time that is specified in the TEDS, the operation times out and returns an error. Otherwise, it grabs the sensor reading, encodes the *ArgumentArray* into the interpretation of the sensor reading, and returns with a successful *OpReturnCode* to the application.

(2) *OpReturnCode WriteAndUpdate(IEEE1451_ParameterWithUpdate *a,*
/ in */ArgumentArray data)*

This operation is used to write a value to an actuator channel and then it triggers the channel. The pseudo-code for this operation is shown in Figure 5-16. From the figure, we can see that after verifying that we are dealing with an actuator channel, we decode the *ArgumentArray* and put it in the format of raw actuator data. Then we trigger the sensor and wait for the Trigger ACK. Again, if the ACK signal is not received within the channel update time that is stored in the TEDS, this operation times out.

```
Check to see what type of transducer we are dealing with
If actuator then
    Decode ArgumentArray and Write results to actuator register
    Trigger the Actuator
    Wait for Trigger ACK

    If timeout then
        Return error message
    Else then
        Return successful operation

Else then
    Print "Tried to write to a Sensor channel so nothing happens"
    Return successful operation
```

Figure 5-16 WriteAndUpdate() Pseudo-code

Next, we discuss the Physical Parameter class since this class is used to access the Channel-TEDS information. The generated structure for this class is shown in Figure 5-17.

```
typedef struct {
    IEEE1451_ParameterWithUpdate *super;
    ClassID class_ID;
    IEEE1451_String description;
    IEEE1451_String parent_name;
    ObjectTag object_tag;
    ObjectID object_ID;
    unsigned short block_state : 8;
    PhysicalParameterType physical_type;
    ParameterInterpretation interpretation;
}IEEE1451_PhysicalParameter;
```

Figure 5-17 Physical Parameter Class Structure in C

Note that most of the parameters that are defined for this class are the same as the other classes that we have defined so far. However, we define a *physical_type* and *interpretation* parameters. These parameters are enumerations that are used to determine the type of transducer (*physical_type*) and the interpretation of the transducer data (sensor reading, actuator value, etc).

(1) *OpReturnCode GetPhysicalParameterType(IEEE1451_PhysicalParameter *a,*
/ out */ unsigned short *parameter_type)*

This operation returns the *physical_type* parameter of the Physical Parameter object.

(2) *OpReturnCode GetMetadata(IEEE1451_PhysicalParameter *a,*
/ out */ PhysicalParameterMetadata *metadata)*

This operation returns the complete structure of the Channel-TEDS of the Physical Parameter object that is being accessed.

(3) *OpReturnCode GetInterpretation(IEEE1451_PhysicalParameter *a,*
*unsigned short *parameter_interpretation,*
*unsigned short *buffering)*

In this operation, the *parameter_interpretation* returns the *interpretation* parameter of the class, and the *buffering* type returns a zero since there is no queuing of Parameter values.

5.1.4 Application Layer

This layer consists of the different NCAP objects that remain unchanged and do not provide hooks for communications with a network or TIM. In this section, we will focus on the design of those objects that control the NCAP's functionality (Block and NCAP Block classes), and the application-specific functionality (given by the Function Block).

5.1.4.1 Block Class

This class is the root hierarchy of all Block objects. To represent this class, we use a structure that has the attributes of an IEEE 1451.1 class header as well as particular information that is pertinent to this object. The generated structure for this object class is shown in Figure 5-18.

The fields shown in the figure are further explained in the following discussion. The *block_state* field is used to represent the current state of the state machine for this class.

The *group_ids* field is an array of chars (*OctetArray*) that represent the sets of objects of which this Block instance is a member.

The *server_object_properties* field is an array of *ObjectProperties* that is used to store information about the server object of the NCAP.

```
typedef struct {
    IEEE1451_Entity *super;
    ClassID class_ID;
    IEEE1451_String description;
    IEEE1451_String parent_name;
    ObjectTag object_tag;
    ObjectID object_ID;
    unsigned short block_state : 8;
    char *group_ids;
    ObjectPropertiesArray *server_object_properties;
}IEEE1451_Block;
```

Figure 5-18 Block Class Structure in C

Using this structure, we can now begin to define the different operations that were previously shown in the specifications chapter. In the following discussion we will design the operation, its signature and an explanation of its design and behavior.

(1) *OpReturnCode GetGroupIDs (IEEE1451_Block *a,*
/ out */ OctetArray group_ids)*

This operation returns the attribute *group_ids* of the Block class.

(2) *OpReturnCode SetGroupIDs (IEEE1451_Block *a,*
/ in */ OctetArray group_ids)*

This operation sets the attribute *group_ids* of the Block class. Note that this operation is not operational when the Block is in the BL_ACTIVE state, so if it is called when in this state the operation returns an error message.

(3) *OpReturnCode GetBlockMajorState (IEEE1451_Block *a,*
/ out */ unsigned short block_major_state)*

This operation returns the current state of the state machine for the Block class. Therefore, it returns *block_state* field that was shown in the structure for the Block class.

(4) *OpReturnCode GetBlockManufacturerID (IEEE1451_Block *a,*
/ out */ String block_manufacturer_ID)*

This operation returns the *block_manufacturer_ID* String with every field set to zero.

(5) *OpReturnCode GetBlockModelNumber (IEEE1451_Block *a,*
/ out */ String block_model_number)*

This operation returns the *block_model_number* String with every field set to zero.

(6) *OpReturnCode GetBlockVersion (IEEE1451_Block *a,*
/ out */ String block_software_version)*

This operation returns the *block_software_version* String with every field set to zero.

(7) *OpReturnCode GoActive (IEEE1451_Block *a)*

This operation causes a transition in the state machine for this block. The pseudo-code for this operation is shown in Figure 5-19.

```

Check the current value of the block_state

If Block is in BL_UNINITIALIZED state then
    Transition to BL_INACTIVE state
    Return successful operation

Else if Block is in BL_INACTIVE state then
    Remain in this state
    Return successful operation

Else Block is in BL_ACTIVE state
    Return error, function not operational

```

Figure 5-19 GoActive() operation pseudo-code

(8) *OpReturnCode GoInactive (IEEE1451_Block *a)*

This operation is used for transitions within the state machine for this Block object. The pseudo-code for this operation is shown in Figure 5-20.

```

Check the current value of the block_state

If Block is in BL_UNINITIALIZED state then
    Return error, function not operational

Else if Block is in BL_INACTIVE state then
    Remain in this state
    Return successful operation

Else Block is in BL_ACTIVE state
    Transition to BL_INACTIVE state
    Return successful operation

```

Figure 5-20 GoInactive() operation pseudo-code

(9) *OpReturnCode Initialize (IEEE1451_Block *a)*

This operation is used for transitions within the state machine for this Block object. The pseudo-code for this operation is shown in Figure 5-21.

```

Check the current value of the block_state

If Block is in BL_UNINITIALIZED state then
    Return error, function not operational

Else if Block is in BL_INACTIVE state then
    Transition to BL_ACTIVE state
    Return successful operation

Else Block is in BL_ACTIVE state
    Remain in this state
    Return successful operation

```

Figure 5-21 Initialize() operation pseudo-code

(10) *OpReturnCode Reset (IEEE1451_Block *a)*

This operation makes the state machine of the Block object to transition to the BL_UNINITIALIZED state regardless of the state that is in. Therefore, the code for this operation consists of setting the *block_state* parameter to the BL_UNINITIALIZED state.

(11) *OpReturnCode GetNetworkVisibleServerObjectProperties (IEEE1451_Block *a,*
/ out */ ObjectTag this_block_object_tag,*
/ out */ ObjectPropertiesArray server_object_properties)*

This operation returns the *object_tag* and *server_object_properties* parameters of the Block object.

5.1.4.2 NCAP Block Class

The structure that is generated for this block is shown in Figure 5-22. Note that most of the parameters that are defined for this class are similar to the ones of the Block class and have the same meaning. Therefore, we will only explain the parameters that are specific to this object class.

```
typedef struct {
    IEEE1451_Block *super;
    ClassID class_ID;
    IEEE1451_String description;
    IEEE1451_String parent_name;
    ObjectTag object_tag;
    ObjectID object_ID;
    unsigned short *block_state;
    unsigned short sub_state : 8;
    ClientPortPropertiesArray *client_port_properties;
    Boolean ignore_request;
}IEEE1451_NCAPBlock;
```

Figure 5-22 NCAP Block Structure in C

The *block_state* is a pointer to the state machine of the Block class. This is done because the operations of this class still depend on the value of that state machine.

The *sub_state* field is used to represent to current value of the sub-stated state machine of the NCAP Block class. Recall that this class substates the BL_ACTIVE state into two states which are represented by this field.

The *client_port_properties* field is an array of *ClientPortProperties* that is used to store information about the client object of the NCAP.

The *ignore_request* field is used to determine if the NCAP Block should respond to the publication. If it is TRUE then we should ignore the publication.

Next, we will design the different operations that are defined for the NCAP Block along with a brief explanation of their pseudo-code.

```
(1) OpReturnCode GetNCAPBlockState(IEEE1451_NCAPBlock *a,
/* out */ unsigned short *ncap_block_state)
```

This operation returns the current value of the sub-state of the state machine for the NCAP Block.

```
(2) OpReturnCode GetNCAPManufacturerID(IEEE1451_NCAPBlock *a,
/* out */ IEEE1451_String *ncap_manufacturer_id)
```

This operation returns the *ncap_manufacturer_id* String with every field set to zero.

(3) *OpReturnCode GetNCAPModelNumber(IEEE1451_NCAPBlock *a,*
/ out */ IEEE1451_String *ncap_model_number)*

This operation returns the *ncap_model_number* String with every field set to zero.

(4) *OpReturnCode GetNCAPSerialNumber(IEEE1451_NCAPBlock *a,*
/ out */ IEEE1451_String *ncap_serial_number)*

This operation returns the *ncap_serial_number* String with every field set to zero.

(5) *OpReturnCode GetNCAPOSVersion(IEEE1451_NCAPBlock *a,*
/ out */ IEEE1451_String *ncap_os_version)*

This operation returns *ncap_os_version* String with every field set to zero.

(6) *OpReturnCode GetClientPortProperties(IEEE1451_NCAPBlock *a,*
/ out */ ObjectTag *this_ncap_block_object_tag,*
/ out */ ClientPortPropertiesArray *client_port_properties)*

This operation returns the *object_tag* and *client_port_properties* parameters of the NCAP Block class.

(7) *OpReturnCode SetClientPortServerObjectBindings(IEEE1451_NCAPBlock *a,*
/ in */ ObjectTag this_ncap_block_object_tag,*
/ in */ ObjectPropertiesArray client_port_server_properties)*

This operation initializes/modifies the *object_tag* and *client_port_properties* parameters of the NCAP Block class.

(8) *OpReturnCode IgnoreRequestNCAPBlockAnnouncement(NCAPBlock *a)*

This operation sets the *ignore_request* parameter of this class to TRUE.

(9) *OpReturnCode RespondToRequestNCAPBlockAnnouncement(NCAPBlock*a)*

This operation sets the *ignore_request* parameter of this class to FALSE.

(10) *OpReturnCode GetBlockCookie(IEEE1451_NCAPBlock *a,*

/ in */ IEEE1451_Object block_reference,*

/ out */ unsigned short *block_cookie)*

In this operation the *block_cookie* of the object that is being referenced is returned.

(11) *OpReturnCode RebootNCAPBlock(IEEE1451_NCAPBlock *a)*

In this operation the NCAP Block and all objects that are owned by this object are placed in the uninitialized state. Therefore, the code for this operation consists of recursively calling all the objects that are owned by the NCAP Block and setting the value of their state machine to the uninitialized state.

(12) *OpReturnCode ResetOwnedBlocks(IEEE1451_NCAPBlock *a)*

In this operation, all blocks that are owned by the NCAP Block behave as if they received a reset. Therefore, the code for this operation consists of recursively calling all the objects that are owned by the NCAP Block and setting the value of their state machine to the uninitialized state.

(13) *OpReturnCode PSK_NCAPBLOCK_GO_ACTIVE (NCAPBlock *a)*

This operation has the same behavior as the *GoActive* operation of the Block class. So it has the pseudo-code that was shown in Figure 5-20.

(14) *OpReturnCode GoInactive (IEEE1451_NCAPBlock *a)*

This is the inherited operation from the Block class, so it has the same behavior as the *GoInactive* operation of the Block class.

5.1.4.3 Function Block Class

The generated structure for this class is shown in Figure 5-23. The parameters for this class have the same meaning as the ones that have been defined for the previous classes.

```
typedef struct {  
    IEEE1451_Block *super;  
    ClassID class_ID;  
    IEEE1451_String description;  
    IEEE1451_String parent_name;  
    ObjectTag object_tag;  
    ObjectID object_ID;  
    unsigned short *block_state;  
    unsigned short sub_state : 8;  
}IEEE1451_FunctionBlock;
```

Figure 5-23 Function Block Class Structure in C

The different operations that are defined for this class are further discussed in the following discussion.

(1) *OpReturnCode GetFunctionBlockState(IEEE1451_FunctionBlock *a,*
/ out */ unsigned short *function_block_state)*

This operation returns the *sub_state* parameter of the Function Block object.

(2) *OpReturnCode Start(IEEE1451_FunctionBlock *a)*

This operation is used for transitions within the sub-stated BL_ACTIVE state of the Function Block. The pseudo-code for this operation is shown in Figure 5-24.

```

Check the Block state
If Block is in BL_ACTIVE state then
    If Function Block is in FB_IDLE state then
        Transition to FB_RUNNING state
        Return successful operation
    Else if Function Block is in FB_RUNNING state then
        Remain in this state
        Return successful operation
    Else Function Block is in FB_STOPPED state then
        Return Error, function not operational
Else Block is in BL_INACTIVE or BL_UNINITIALIZED state then
    Return Error, function not operational

```

Figure 5-24 Start() Operation pseudo-code

(3) *OpReturnCode Clear(IEEE1451_FunctionBlock *a)*

This operation is used for transitions within the sub-stated BL_ACTIVE state of the Function Block. The pseudo-code for this operation is shown in Figure 5-25.

```

Check the Block state
If Block is in BL_ACTIVE state then
    If Function Block is in FB_IDLE state then
        Remain in this state
        Return successful operation
    Else if Function Block is in FB_RUNNING state then
        Transition to FB_IDLE state
        Return successful operation
    Else Function Block is in FB_STOPPED state then
        Return Error, function not operational
Else Block is in BL_INACTIVE or BL_UNINITIALIZED state then
    Return Error, function not operational

```

Figure 5-25 Clear() Operation pseudo-code

(4) *OpReturnCode Pause(IEEE1451_FunctionBlock *a)*

This operation is used for transitions within the sub-stated BL_ACTIVE state of the Function Block. The pseudo-code for this operation is shown in Figure 5-26.

```

Check the Block state

If Block is in BL_ACTIVE state then

    If Function Block is in FB_STOPPED state then
        Remain in this state
        Return successful operation

    Else if Function Block is in FB_RUNNING state then
        Transition to FB_STOPPED state
        Return successful operation

    Else Function Block is in FB_IDLE state then
        Return Error, function not operational

Else Block is in BL_INACTIVE or BL_UNINITIALIZED state then
    Return Error, function not operational

```

Figure 5-26 Pause() Operation pseudo-code

(5) *OpReturnCode Resume(IEEE1451_FunctionBlock *a)*

This operation is used for transitions within the sub-stated BL_ACTIVE state of the Function Block. The pseudo-code for this operation is shown in Figure 5-27.

```

Check the Block state

If Block is in BL_ACTIVE state then

    If Function Block is in FB_STOPPED state then
        Transition to FB_RUNNING state
        Return successful operation

    Else if Function Block is in FB_RUNNING state then
        Remain in this state
        Return successful operation

    Else Function Block is in FB_IDLE state then
        Return Error, function not operational

Else Block is in BL_INACTIVE or BL_UNINITIALIZED state then
    Return Error, function not operational

```

Figure 5-27 Resume() Operation pseudo-code

5.1.5 Summary and Example implementation

In this section, we have shown the complete design of the information model that is defined in the IEEE 1451.1 standard. By implementing this model, we achieve compliance with the specifications that were derived in Section 4.1. Therefore, users can configure this NCAP implementation for their particular application. Next, we will show the configuration for an NCAP that is interfaced with an 8-Channel TIM. For the purpose of this example we will assume that there are four sensors and four actuators implemented in the TIM. Note that we will focus on the NCAP objects that deal with TIM communications.

The first thing that we must do is to instantiate an NCAP Block class. In order to instantiate this object in C we use the scheme that was previously designed in Figure 5-3. Therefore, we write the following statement in the main code:

```
IEEE1451_NCAPBlock *control;
```

Then we would have to initialize the different parameters (previously shown in Figure 5-22) within this structure by using the arrow C operator (e.g. *control->object_tag* = 0). This NCAP Block object consolidates the system and communication housekeeping. Therefore, this class will control the behavior of the experiment. Next, we instantiate (using the same scheme that was shown for the NCAP Block) a Transducer Block so that the TIM transducers can be mapped to NCAP objects. This block is then used to obtain the number of transducers that are implemented within the TIM by accessing the Meta-TEDS through the *GetNumberOfTransducersChannels* shown in the Section 5.1.3.1. From the result of this operation (eight in this example), we proceed to instantiate eight parameter objects that are used to represent the individual channels. In order to do this instantiation we can use C *malloc()* operation in the main program code as follows:

```
IEEE1451_Parameter *parameters= (IEEE1451_Parameter *)malloc(8*sizeof(IEEE1451_Parameter));
```

By doing this, we have mapped all the TIM transducers to NCAP objects, so the next step is to define any application-specific behavior of the system. For the purpose of this example, we assume that the transducers need to be sampled/set every two seconds with n number of iterations. In order to accomplish this, we instantiate a Function Block that has an application-specific operation called *SampleSetAll()*. This operation calls the *ReadAndUpdate* and *WriteAndUpdate* operations (used to trigger the transducers as was previously shown in Section 5.1.3.2) of the transducers every two seconds for the number of iterations that the application requires. This application-specific operation is shown in Figure 5-28. It is important to note that the *timer* variable that is shown in the code is an interrupt driven signal that increments every second.

```
SampleSetAll(IEEE1451_FunctionBlock *a, int iterations){
    int count = 0;
    While(count <= iterations){
        UpdateAndRead(sensor1, data1);
        UpdateAndRead(sensor2, data2);
        UpdateAndRead(sensor3, data3);
        UpdateAndRead(sensor4, data4);
        WriteAndUpdate(actuator1, writeData1);
        WriteAndUpdate(actuator2, writeData2);
        WriteAndUpdate(actuator3, writeData3);
        WriteAndUpdate(actuator4, writeData4);
        // Reset the timer...
        timer = 0;
        // Wait until timer reaches two seconds
        While (timer <= 2);
        count++;
    }
    return 0;
}
```

Figure 5-28 Application-specific example operation SampleAndSetAll()

Next, we design the NCAP/TIM commands that are sent through the physical interface (AMBA AHB). Because of the memory mapping that exists within the Excalibur chip, we make the design decision to implement the commands as a memory address. In doing this, we need to

assign the memory addresses for the mandatory TIM commands. These mandatory commands include the following:

- The *TIM_Channel_MASK* command is used to issue read/write to the transducer channel's interrupt mask register.
- The *TIM_Channel_STATUS* command is used to read the transducer channel's status register. Note that this command is read-only.
- The *TIM_Channel_TRIGGER* command is used to trigger the selected transducer channel. Note that this command is asserted when this address space is accessed regardless of it is a read/write access.
- The *TIM_Channel_DATA* command is used to read/write the transducer channel's data register. Note that for sensors this command is read-only, while it is write-only for actuators.
- The *TIM_RESET* command is used to reset the TIM, which places all the TIM's objects in their default power-on state.

Next, we need to design the memory address spaces such that we have enough space for a future expansion in which up to 255 transducers can be accessed. In order to do this, we can configure the TIM to have a base address of 0x80000000 with a range of 64KB. Then, the 16-bit offset address is split into channel and functional addresses. Then, in order to leave to be able to represent 255 transducers, we use the least significant bits of the offset address for the channel address, while the remaining most significant bits of the 16-bit offset address are used to represent the functional address (command). With this in mind, we can design a header file for this 8-Channel TIM example in Figure 5-29.

Note that to access each of these address spaces, all that we have to do is to define the base address of the TIM and read/write to it depending on the command. For example, if we want to write an 0xFF (HEX notation) value to the interrupt mask register of channel one, we write `*TIM_ONE_MASK(EXC_PLD_BLOCK0_BASE) = 0xFF`. Note that the base address is EXC_PLD_BLOCK0_BASE, which is a constant that we can define in a header file or main code.

Using the header file that was previously shown in Figure 5-29, we can show how to customize the *UpdateAndRead* and *WriteAndUpdate* operations for a particular application. For the purpose of the example we assume that channel one is a temperature sensor, and channel three is an actuator. With this in mind, we can show the snippet C code that is needed for the *UpdateAndRead* operation when it is called for the temperature sensor in Figure 5-30.

```
#ifndef TIM_CTRL00_H
#define TIM_CTRL00_H

#define EXC_PLD_TYPE (volatile unsigned int *)

// Define all the interrupt masks
#define TIM_ZERO_MASK(base_addr) (EXC_PLD_TYPE (base_addr + 0x0000))
#define TIM_ONE_MASK(base_addr) (EXC_PLD_TYPE (base_addr + 0x0004))
#define TIM_TWO_MASK(base_addr) (EXC_PLD_TYPE (base_addr + 0x0008))
...
#define TIM_EIGHT_MASK(base_addr) (EXC_PLD_TYPE (base_addr + 0x0020))

// Define the status registers
#define TIM_ZERO_STATUS(base_addr) (EXC_PLD_TYPE (base_addr + 0x2000))
#define TIM_ONE_STATUS(base_addr) (EXC_PLD_TYPE (base_addr + 0x2004))
#define TIM_TWO_STATUS(base_addr) (EXC_PLD_TYPE (base_addr + 0x2008))
...
#define TIM_EIGHT_STATUS(base_addr) (EXC_PLD_TYPE (base_addr + 0x2020))

// Trigger...
#define TIM_ZERO_TRIGGER(base_addr) (EXC_PLD_TYPE (base_addr + 0x4000))
#define TIM_ONE_TRIGGER(base_addr) (EXC_PLD_TYPE (base_addr + 0x4004))
#define TIM_TWO_TRIGGER(base_addr) (EXC_PLD_TYPE (base_addr + 0x4008))
...
#define TIM_EIGHT_TRIGGER(base_addr) (EXC_PLD_TYPE (base_addr + 0x4020))

// Data registers...
#define TIM_ZERO_DATA(base_addr) (EXC_PLD_TYPE (base_addr + 0x6000))
#define TIM_ONE_DATA(base_addr) (EXC_PLD_TYPE (base_addr + 0x6004))
#define TIM_TWO_DATA(base_addr) (EXC_PLD_TYPE (base_addr + 0x6008))
...
#define TIM_EIGHT_DATA(base_addr) (EXC_PLD_TYPE (base_addr + 0x6020))

// Reset Command
#define TIM_RESET(base_addr) (EXC_PLD_TYPE (base_addr + 0x8000))

#endif /* TIM header file */
```

Figure 5-29 TIM Commands Header File

The behavior of this operation was written using the pseudo-code that was designed previously in Figure 5-15. Therefore, we trigger the sensor and then we wait until we receive the trigger ACK. If we do not receive this trigger ACK within the time that is specified by the Channel-TEDS, we timeout and return the timeout enumeration that is given by the IEEE 1451.1 standard. If the operation does not timeout, we proceed to read the raw sensor data. This is later converted to Celsius though an application-specific function that has this information. Then the last step is to encode the converted result into an argument array and we return signaling that the operation executed successfully.

```
// Temperature sensor
// Trigger the sensor then read...
// Next Line Changes depending on the transducer channel
*STIM_ONE_TRIGGER(EXC_PLD_BLOCK0_BASE) = 1;

// Timeout routine
while (trigger_ACK != 1) {
    // Trigger_ACK must be received within the channel Update Time specified in the TEDS
    if (jiffies > channel1.ChannelUpdateTime)
        return 14; // Return Operation Timed out
}
// Next Line Changes depending on the transducer channel
// Reading raw sensor data
raw = *STIM_ONE_DATA(EXC_PLD_BLOCK0_BASE);
// Converting temperature to Celsius
ConvertTemperature(raw,res);
// Encoding the Celsius temperature into the output argument array
data->typeCode = FLOAT32_TC;
data->Arg_Union.float32Val = *res;
return 0;
```

Figure 5-30 Snippet C Code for UpdateAndRead Operation

On the other hand, we can also show a snippet of the code that is needed for the behavior of the *WriteAndUpdate* operation when it is called for Channel three in Figure 5-31.

```

//Next two Lines Change depending on the transducer channel
// Writing Data to Actuator Data Register
*STIM_THREE_DATA(EXC_PLD_BLOCK0_BASE) = data->Arg_Union.integer32Val;
// Trigger the Actuator
*STIM_THREE_TRIGGER(EXC_PLD_BLOCK0_BASE) = 1;
// Timeout Routine
while (trigger_ACK != 1) {
    // Trigger_ACK must be received within the channel Update Time specified in the TEDS
    if (jiffies > channel2.ChannelUpdateTime)
        return 14; // Return Operation Timed out
    }
    return 0;
}

```

Figure 5-31 Snippet C Code for WriteAndUpdate Operation

The behavior of this operation was written using the pseudo-code that was designed previously in Figure 5-16. Therefore, we first write the actuator data to the transducer channel's data register. Then, we set the actuator with this data by applying a trigger command. After doing this, we need to wait for the trigger ACK from the TIM to make sure that the operation was executed correctly. Again, if we do not receive this trigger ACK within the specified TEDS time the operation times out.

Lastly, for the network interface, we must instantiate the different objects that are used in client/server and publish/subscribe communications. These objects include the Entity, Client Port, Publisher Port, and Subscriber Port classes. It is important to note that this mapping needs to be done at compile-time since this NCAP design does not support dynamic downloading.

Note that the configuration that we have shown in this example is the minimal configuration for an IEEE 1451.1 NCAP. Therefore, if the application requires more objects then it is the designer's responsibility to implement them in such that they fit the standard, and can be *plugged* into the NCAP.

5.2 TRANSDUCER INTERFACE MODULE

The TIM is designed using a combination of hardware and software in a way that different transducers can be added or deleted with no major effort. In doing this, we allow easy portability from the EPXA1 chip to the EPXA10 chip. This porting is simple because these chips both have the same architecture, so all we have to do is design the standard-defined blocks in a generic way so that they can be reused with little effort. The architecture shown in Figure 5-32 allows for this level of configurability.

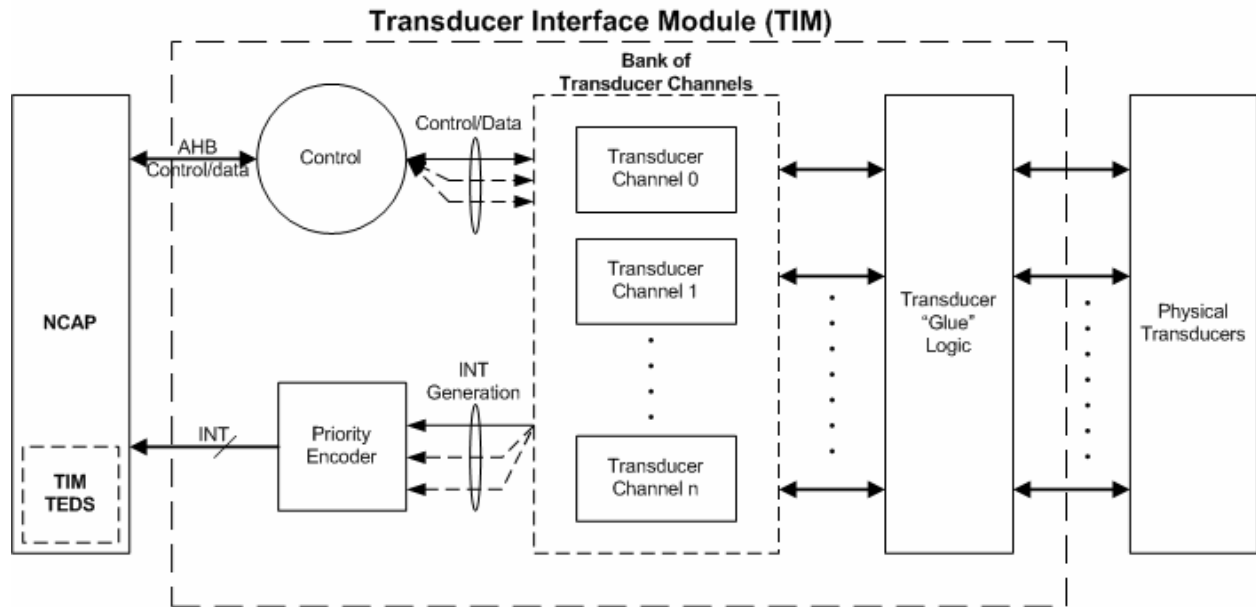


Figure 5-32 TIM high-level Architecture

Note that for the transducer channels, we design a generic block that contains three registers. This block interfaces with the control, priority encoder, and "glue" logic blocks as is shown in the figure. The registers within this block consist of the transducer data (sensor read or actuator write), the status register, and the interrupt mask register. Also, within this block we perform the AND operation between the status and interrupt mask registers, since this is the scheme (previously described in Section 4.2.4) that is used to generate an interrupt. The output of

this operation is sent to the priority encoder. This encoder generates an interrupt signal that is interfaced with an interrupt controller. The interface between each transducer channel block and the control unit consists of control and data signals that are used to read/write the registers within a transducer channel. Also, each transducer channel block also interfaces with “glue” logic that is specific to the transducer that is being interfaced to the TIM. Lastly, the TEDS is designed in software as part of the NCAP as will be shown in the following discussion.

5.2.1 Transducer Electronic Data Sheet

There are two options to design the TEDS such that it resides in non-volatile memory and is accessible to the NCAP. The first option is to implement it in a custom Read Only Memory (ROM) block within the PLD.

The second option is for the TEDS to reside within the NCAP and be designed in software while storing it in the Flash memory that is used for the program memory. This option gives a high level of configurability. This is because the NCAP would be able to access the TEDS structure by the operations that are pre-defined in the NCAP’s object model. So, there would be no need for a command to be sent over the AHB. The high level of configurability and the easy-access of the block by the NCAP make a software-based TEDS the best option.

For the software-based TEDS we create a header file. This header file consists of two structure types (Meta-TEDS and Channel-TEDS) that contain the information previously shown in Table 4-3 and Table 4-4. This way, the TEDS information is set during compile-time with the values of the TIM and its transducers. Users can access this information through the operations that are defined for the NCAP.

5.2.2 Transducer Interface Module Control Unit

The TIM control unit consists of two interfaces. It has an interface that is used for AHB communications and it has another interface for the transducer channels. The AHB portion of this unit is fixed and is designed to meet the timing specifications of the bus. On the other hand, for the transducer interface, users have to define all of the control/data signals that need to be interfaced with the transducer channel. This is done because it is impossible to dynamically set the port size of a VHDL entity, and the number of signals that are interfaced to the transducer channels varies depending on the amount of transducers that are implemented. Therefore, the designer has to define these signals at compile-time so that this block can interface with the transducer blocks.

Next, we describe the AHB communication portion of the controller. To allow the NCAP to control the TIM as is required by the standard, we design the TIM control unit as a slave to the embedded stripe. Therefore, the control unit needs to comply with the timing diagrams of the bus. This timing is shown in Figure 5-33.

Note that transactions on this bus consist of two distinct phases. An address phase, which lasts only a single cycle and a data phase that may require several cycles. The data phase is controlled by the HREADY signal. The events that are shown in the figure can be further explained by the following discussion. The master drives the address and control signals onto the bus after the rising edge of HCLK. Then, the Slave samples the address and control information on the next rising edge of the clock. After the slave has sampled the address and control it can start to drive the appropriate response and this is sampled by the bus master. Next, we define the signal structure for the protocol. This structure is summarized in Figure 5-34.

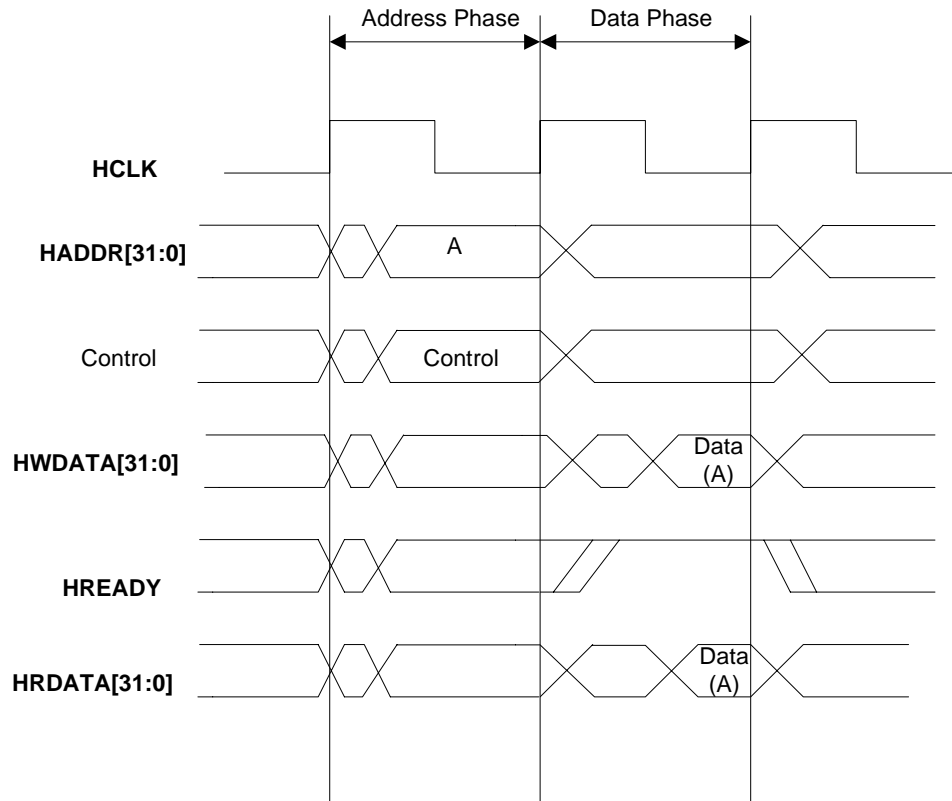


Figure 5-33 Simple AMBA transfer ⁽¹⁹⁾

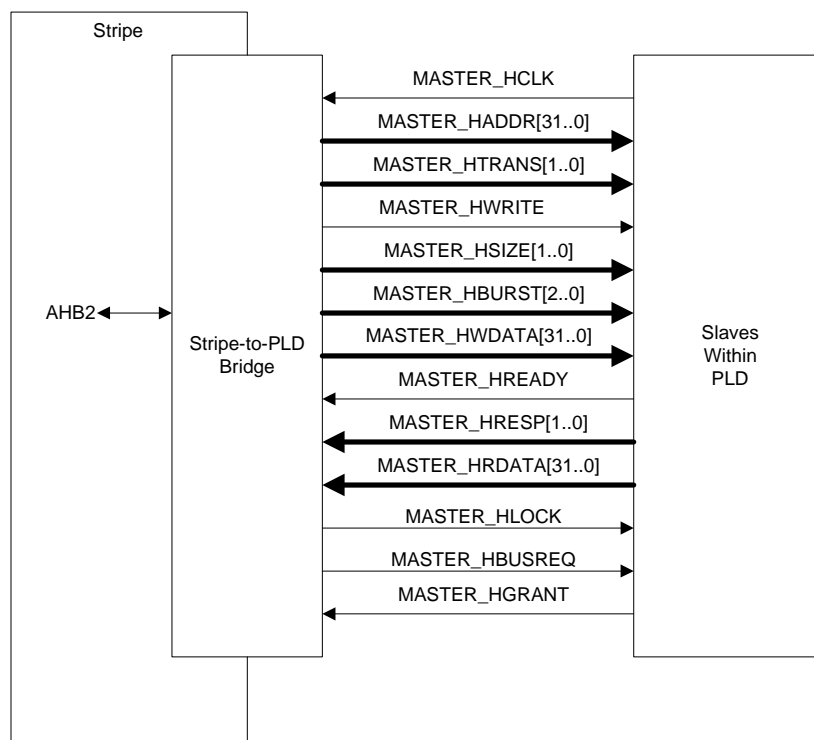


Figure 5-34 PLD Slave and AHB Connection

The different bus communication signals that are shown in the figure are explained in Table 5-2. It is important to note that in addition to the signals that are shown in the table, each slave also has a select signal (HSEL) that indicates that the current transfer is intended for the selected slave. This signal is a combinatorial decode of the address bus signal HADDR.

Table 5-2 AHB Signal Summary

Signal	Source	Description
MASTER_HCLK	PLD	Stripe-to-PLD bridge slave-port clock that times all bus transfers. Signal timings are related to its rising edge clock. This signal is invertible
MASTER_HADDR[31..0]	Stripe	Stripe-to-PLD 32-bit system address bus
MASTER_HTRANS[1..0]	Stripe	Stripe-to-PLD bridge transfer type. 00 IDLE, 01 BUSY, 10 NONSEQ, 11 SEQ
MASTER_HWRITE	Stripe	When high, indicates a write transfer on the stripe-to-PLD bridge; when low, a read transfer
MASTER_HSIZE[1..0]	Stripe	Indicates the size of transfer on the stripe-to-PLD bridge. 00 BYTE, 01 Half word, 10 Word, 10 Double word
MASTER_HBURST[2..0]	Stripe	Indicates whether the stripe-to-PLD bridge transfer forms part of a burst.
MASTER_HWDATA[31..0]	Stripe	Used to transfer data from the master to the bus slaves during writes across the stripe-to-PLD-bridge
MASTER_HREADY	PLD	When high, indicates that a stripe-to-PLD bridge transfer has finished
MASTER_HRESP[1..0]	PLD	Slave response that provides additional information on the status of a transfer across the stripe-to-PLD bridge
MASTER_HRDATA[31..0]	PLD	Used to transfer data from the bus slaves to the master during reads across the stripe-to-PLD bridge
MASTER_HLOCK	Stripe	When high, indicates that the master requires locked access to the bus
MASTER_HBUSREQ	Stripe	Bus request signal, from the master to the arbiter
MASTER_HGRANT	PLD	Bus grant signal that, in conjunction with MASTER_HREADY, indicates that the bus master has been granted the bus

To meet the timing requirements of the AHB (previously shown in Figure 5-33), the behavior of the control unit is controlled by a state machine. This state machine is shown in Figure 5-35.

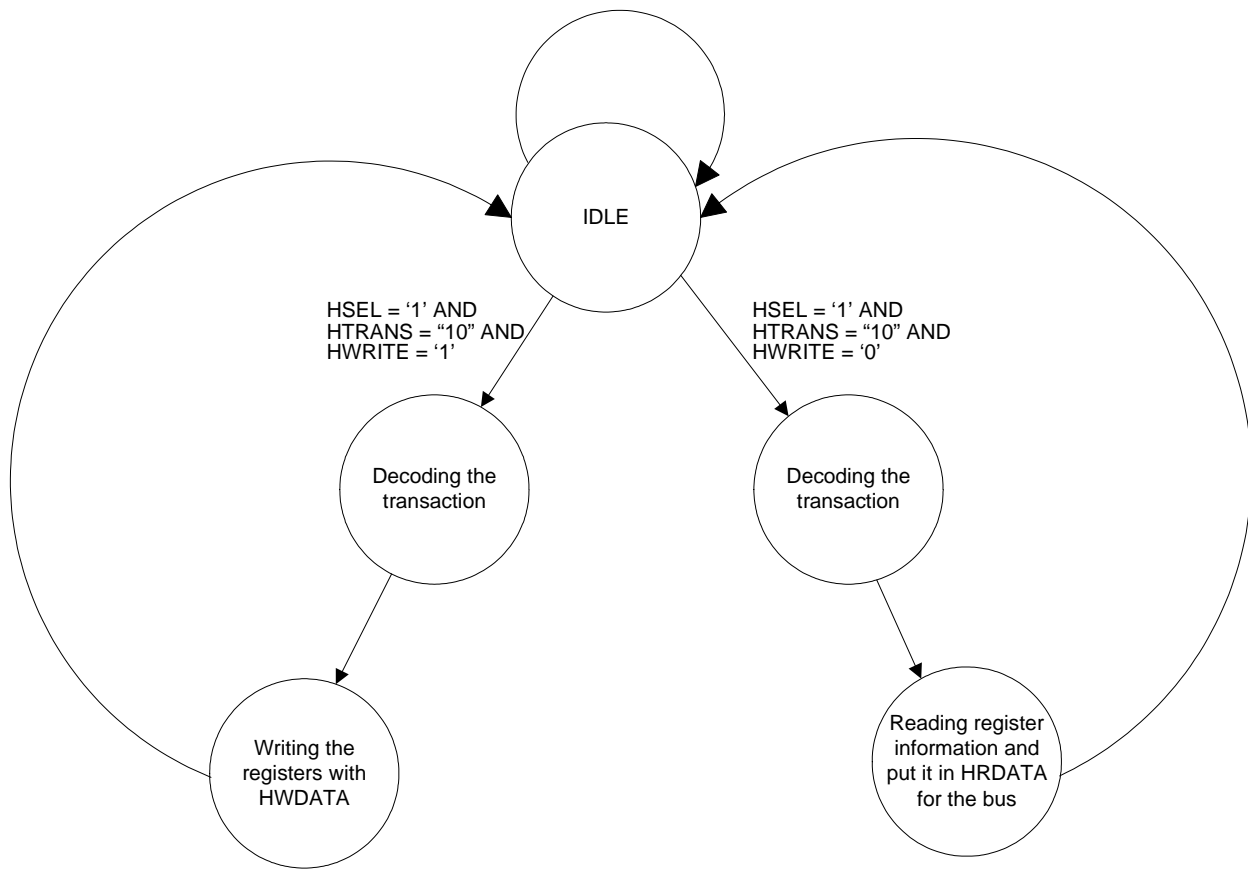


Figure 5-35 State Machine for the Control Unit

Following the timing protocol of the bus, we design the controller state machine to remain idle until it is selected for a bus transfer by the NCAP (e.g. NCAP sends a Trigger command). Then, there are two basic subroutines one for read and the other for write operations. When the TIM is selected for a transfer, the controller decodes the command during the address phase of the protocol. For read operations, we read the register information (from the channel data, status, or interrupt mask register of the transducer channel) and place the data on the bus through the HRDATA signal. For write operations, the data that is received from the bus through the HWDATA signal is used to set the register that it is addressing.

Next, we design the pseudo-code for the TIM control unit. This is shown in Figure 5-36. Note that we design the TIM to be selected when the HSEL signal is asserted and the HTRANS

bits are “10” in binary. The HSEL signal is generated by an address decoder, and when asserted it means that the peripheral has been selected for a bus transaction. To make sure that there is going to be a transfer on the bus, we also verify that the transaction signal is not idle or busy. After we have verified that the TIM is in fact selected for a transfer, we decode the command and set the different control signals of the transducer channel. It is important to note that these signals will be asserted when the NCAP sends a command such as *Write Interrupt Mask Register* to the TIM through the AHB.

```

When in Address phase

If HSEL = '1' AND HTRANS = "10" THEN

    TIM is selected
    Check to see if it is a read or write

    If HWRITE = '1' THEN

        Selected for write
        Decode the address and set control signals
        Transition to Data phase

    ELSE HWRITE = '0' THEN
        Selected for read
        Decode the address and set control signals
        Transition to Data phase

Else
    TIM is not selected so stay idle

When in Data phase

    If TIM was selected for write THEN
        Send HWDATA to registers
        Set HREADY and HRESP signals

    ELSE TIM was selected for read THEN
        Place register data in HRDATA of the bus
        Set HREADY and HRESP signals

```

Figure 5-36 Pseudo VHDL code for TIM Control Unit

Next, we design the transducer-side communications for the control unit. In order to do this we first need to design the command set that was previously derived in Section 4.2.2. To do this, we take advantage of the memory mapping that the Excalibur system's architecture provides. In doing so, we design the TIM's commands as an address (HADDR signal) within the PLD. This improves performance since a command is executed in two bus cycles (address and data phases) at a clock rate that can run as fast as 80 MHz.

In order to fully represent the maximum number of transducers (255) and the commands, we design the PLD base address to be 0x80000000 with a range of 64KB. Then, the 16-bit offset address is split into channel and functional addresses. This is done to represent 255 transducers and the mandatory TIM commands (*Read/Write Interrupt Mask Register*, *Read Status Register*, *Individual/Global Trigger*, *Read/Write Transducer Channel Data*, and *TIM Reset*). To do this, we use the least significant bits of the offset address for the channel address, while the remaining most significant bits of the 64KB are used to represent the functional address (command). Note that to allow for the maximum amount of possible transducers we reserve 10 bits. This is because the memory range is byte-addressable, so it cannot be represented by 8 bits. Rather, channel 255 would have an offset address of 0x3FC (HEX notation), which requires 10 bits for its representation.

From the timing diagram of the bus (previously shown in Figure 5-33), we know that there is a signal HWRITE (previously described in Table 5-2) that denotes the direction of the transfer (read or write). Therefore, the commands that have both read and write capabilities like the interrupt mask can be implemented as a single address. Next, we design the offset memory locations for the TIM commands in Table 5-3.

Table 5-3 AMBA AHB Command Set

Channel	INT_MASK	Status	Trigger	Channel_Data	Reset
ZERO	0x0000	0x2000	0x4000	0x6000	0x8000
1	0x0004	0x2004	0x4004	0x6004	NA
2	0x0008	0x2008	0x4008	0x6008	NA
3	0x000C	0x200C	0x400C	0x600C	NA
...
255	0x03FC	0x23FC	0x43FC	0x63FC	NA

The *INT_MASK* corresponds to the *Read Interrupt Mask* and *Write Interrupt Mask* commands that were discussed in the specifications chapter.

The *Status* command is used to read the status registers of the TIM and its individual transducers. Note that this command is read-only so if we try to write to that memory location it will have no effect.

The *Trigger* command is asserted when the NCAP accesses the memory address shown in the table. When this command is sent, the control unit sends a signal to the “glue” logic. The “glue” logic then configures the different signals that are needed to sample/set the transducer.

The *Channel Data* command is used to read or write to or from the transducer data registers. For sensors, this command is read-only and returns the sensor reading from the last trigger. For actuators, this command is write-only and is used to pre-load the transducer register with the data that it will acquire when it receives the next trigger.

The *Reset* command is asserted when the memory address shown in the table is accessed. This command causes every block within the TIM to return to their default power-on state. For example, all the registers are cleared. Note that this command is only valid for the TIM as a whole.

5.2.2.1 Example Configuration

In this sub-section, we will design the configuration for the control unit for an 8-Channel TIM using the blocks that were designed in the previous section. In order to configure the control unit, we first need to instantiate the different control/data signals for the transducer channels that are implemented. Then, we add the transducer channels and their commands in the state machine of the control unit. Next, we design the connections that are made between the control unit and the transducer channels and their “glue” logic. We begin with the Write Interrupt Mask Register command. For this command, we need to instantiate nine separate enable signals (for CHANNEL_ZERO through Channel eight) and we can use the same data input to every register. This is because the register for each channel is only enabled when the command is selected for that particular channel. With this in mind, we design the configuration for this command in Figure 5-37.

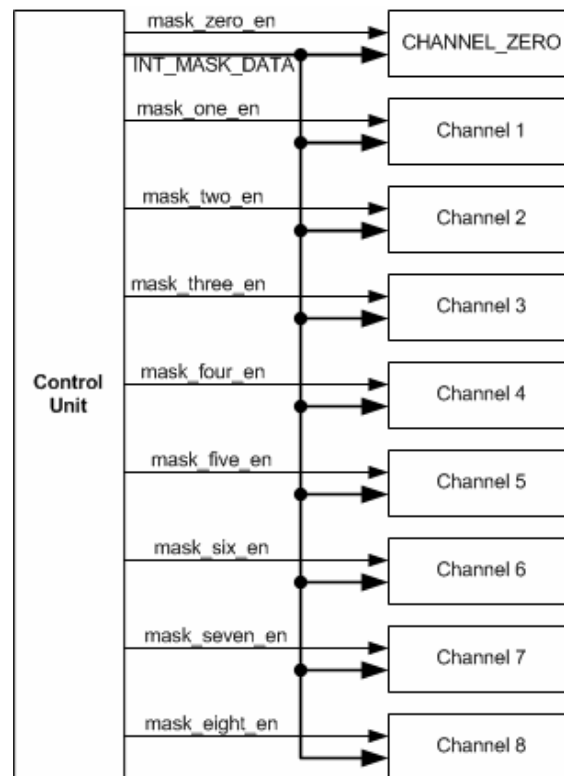


Figure 5-37 Control Unit Example Configuration for Write Interrupt Mask Command

Then, with this configuration we can design the VHDL code that set the enable signals of the different registers as well as place the data that is sent from the NCAP into the interrupt mask register of the channel that is selected. To do this, we follow the pseudo-VHDL code structure that was previously shown in Figure 5-36. Therefore, during the address phase of the bus transaction, we set the enable signals of the registers. Then, in the data phase we place the HWDATA sent from the NCAP through the AHB onto the interrupt mask register's input data. A snippet of the VHDL code that is needed to do this is shown in Figure 5-38.

```
Address Phase
IF HWRITE = '1'
CASE HADDRESS (15 DOWNT0 0) IS
WHEN X"0000" =>
    mask_zero <= '1';
WHEN X"0004" =>
    mask_one <= '1';
WHEN X"0008" =>
    mask_two <= '1';
WHEN X"000C" =>
    mask_three <= '1';
WHEN X"0010" =>
    mask_four <= '1';
WHEN X"0014" =>
    mask_five <= '1';
WHEN X"0018" =>
    mask_six <= '1';
WHEN X"001C" =>
    mask_seven <= '1';
WHEN X"0020" =>
    mask_eight <= '1';

Data Phase
CASE HADDRESS (15 DOWNT0 0) IS
WHEN X"0000" =>
    mask_data <= HWDATA;
WHEN X"0004" =>
    mask_data <= HWDATA;
WHEN X"0008" =>
    mask_data <= HWDATA;
WHEN X"000C" =>
    mask_data <= HWDATA;
WHEN X"0010" =>
    mask_data <= HWDATA;
WHEN X"0014" =>
    mask_data <= HWDATA;
WHEN X"0018" =>
    mask_data <= HWDATA;
WHEN X"001C" =>
    mask_data <= HWDATA;
WHEN X"0020" =>
    mask_data <= HWDATA;
```

Figure 5-38 Snippet VHDL code for TIM Control unit for Write Interrupt Mask Command

Next, we design the configuration for the *Read Interrupt Mask Register* command. The logic design for this command is similar to the one that was previously shown for the Write Interrupt Mask Register command. However, when we read the interrupt mask register we do not need to assert the enable signals of the register. Rather, we use the output information of the interrupt mask register and we place it on the HRDATA signal of the AHB during the data phase. With this in mind, we design the configuration for this command in Figure 5-39. Also, a snippet of the VHDL code is shown in Figure 5-40.

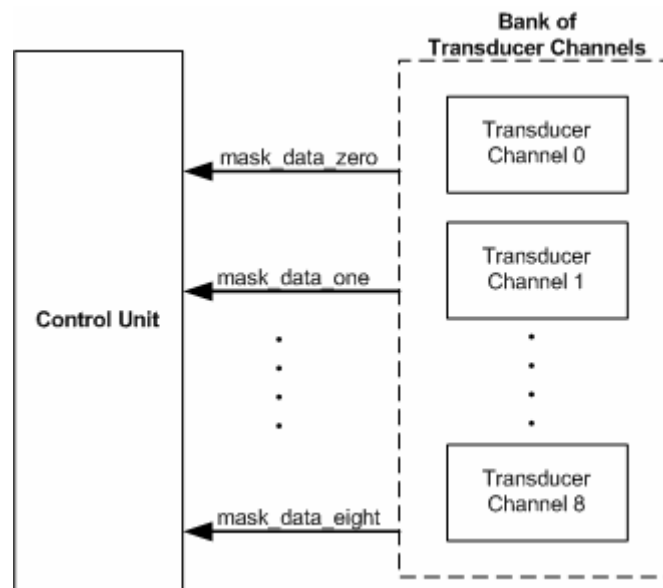


Figure 5-39 Control Unit Example Configuration for Read Interrupt Mask Command

```
Data Phase
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"0000" =>
    HRDATA <= mask_data_zero;
  WHEN X"0004" =>
    HRDATA <= mask_data_one;
  ...
  WHEN X"0020" =>
    HRDATA <= mask_data_eight;
```

Figure 5-40 Snippet VHDL code for TIM Control unit for Read Interrupt Mask Command

Note that we do not show the address phase, since as was mentioned earlier we do not need to set any control signals for this command. Next, we design the configuration for the *Read Status Register* command. This command is read-only, so when it is accessed for a write we need to generate an invalid command condition on CHANNEL_ZERO's status register. Therefore, to signal that the NCAP tried to send an invalid command, we use an invalid signal that is interfaced with the "glue" logic of CHANNEL_ZERO. This is done because the "glue" logic of the transducer channel has the responsibility of generating the control/data bits for the transducer channel's status registers. Also, it is important to note that when an actuator channel's status register is read, the trigger ACK status bit should be cleared. Therefore, we design a signal (*RD_Status*) that is used to denote that the NCAP has requested to read the channel's status register. With this in mind, we design the configuration for this command under the 8-Channel TIM example in Figure 5-41.

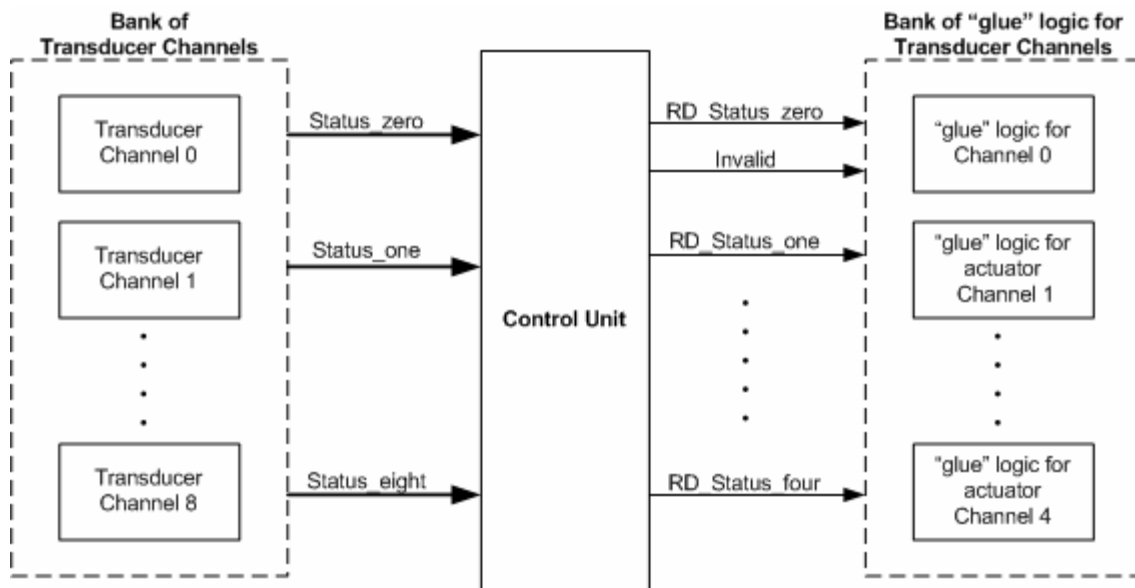


Figure 5-41 Control Unit Example Configuration for Status Command

Note that the *RD_Status* signal is also sent to the "glue" logic of CHANNEL_ZERO. This is because this channel represents the TIM as a whole, so it has both the sensor and actuator

information. Therefore, the trigger ACK status bit for CHANNEL_ZERO's status register is also cleared when the *RD_Status* signal is asserted by the control unit.

Next, we design the VHDL code for this command. During the address phase of this command, we set the control signals (*RD_Status* and *Invalid*) that are interfaced with the “glue” logic of the transducer channels that were previously shown in Figure 5-41. Then, during the data phase we place the output of the status registers (*Status_zero*, *Status_one*, *Status_eight* previously shown in Figure 5-41) on the *HRDATA* signal of the bus. A snippet of the VHDL code for this command is shown in Figure 5-42.

```

Address Phase
IF HWRITE = '0' THEN
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"2000" =>
    RD_Status_zero <= '1';
  WHEN X"2004" =>
    RD_Status_one <= '1';
  ...
  WHEN X"2010" =>
    RD_Status_four <= '1';

ELSE -- HWRITE = '1' THEN
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"2000" =>
    Invalid <= '1';
  WHEN X"2004" =>
    Invalid <= '1';
  ...
  WHEN X"2020" =>
    Invalid <= '1';

Data Phase
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"2000" =>
    HRDATA <= Status_zero;
  WHEN X"2004" =>
    HRDATA <= Status_one;
  ...
  WHEN X"2020" =>
    HRDATA <= Status_eight;

```

Figure 5-42 Snippet VHDL code for TIM Control unit for Status Command

Next, we design the configuration/code for the *Trigger* command. This command is used to sample/set the different transducers that are implemented in the TIM. Note that since the “glue” logic of the transducer channels is responsible for sampling/setting the transducers, we

generate a trigger signal that is interfaced with the “glue” logic. Therefore, when this signal is asserted, the transducer channel’s “glue” logic should sample the sensor and/or set the actuator. Taking this into account, we design the configuration for the Trigger command in Figure 5-43.

Then, using this configuration we design the VHDL code that is used to handle this command. As was mentioned earlier in this section, this command is asserted when the offset address (previously shown in Table 5-3) is accessed regardless of it is a read or write command. Therefore, we assert the trigger signal in the address phase regardless of if it is a read or write request that is sent to the bus. A snippet of the VHDL code for this command is designed in Figure 5-44.

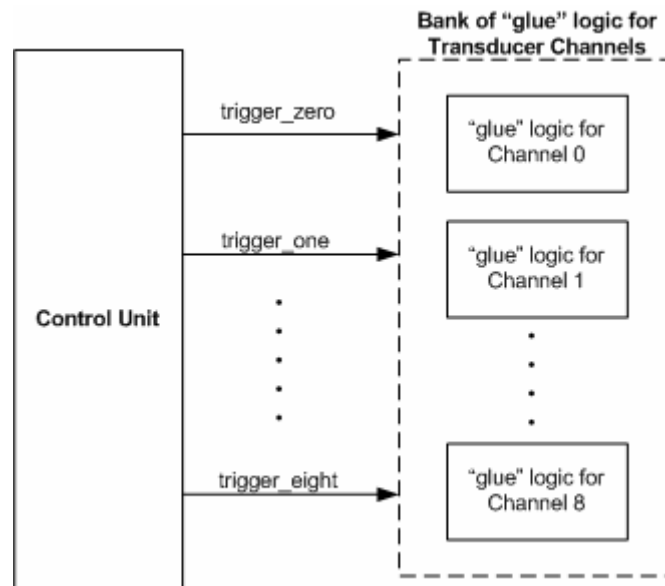


Figure 5-43 Control Unit Example Configuration for Trigger Command

```

Address Phase
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"4000" =>
    trigger_zero <= '1';
    trigger_one <= '1';
    ...
    trigger_eight <= '1';
  WHEN X"4004" =>
    trigger_one <= '1';
    ...
  WHEN X"4020" =>
    trigger_eight <= '1';

```

Figure 5-44 Snippet VHDL code for TIM Control unit for Trigger Command

Next, we design the configuration for the *Channel_Data* command that was previously shown in Table 5-3. When this command is selected for a write, we write the actuator data that will be used upon reception of the next trigger command. Therefore, to write this actuator data (sent by the NCAP through the AHB) we design the control unit to set the control/data signals of the actuator channel's data register. With this in mind, we design the configuration for the Write Actuator Data command in Figure 5-45. It is important to note that the Write Channel Data command is only valid for an actuator channel, so if a sensor channel is selected for this command we ignore the command and nothing occurs.

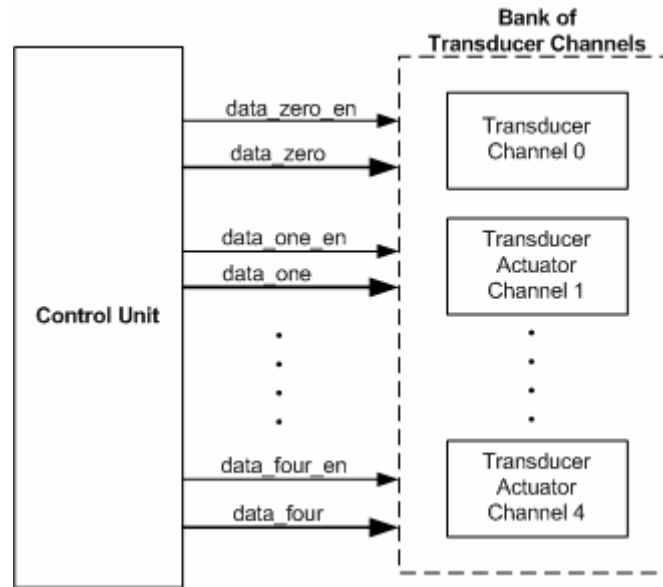


Figure 5-45 Control Unit Example Configuration for Write Actuator Data Command

For the VHDL code of this command, we set the control signals (*data_zero_en*, *data_one_en*, etc.) during the address phase. Then during the data phase we set the input data of the actuator register (*data_zero* through *data_four*) to the HWDATA signal of the bus. A snippet of this VHDL code is shown in Figure 5-46.

On the other hand, when this command is selected for a read, we read sensor data register of the transducer channel block. Note that when this command is asserted, the Trigger ACK bit of the status registers of the sensor channels is cleared. Therefore, we need to generate a signal (*RD_Sensor*) that is interfaced with the sensor channel's "glue" logic, so that the Trigger ACK status bit can be cleared appropriately. With this in mind, we design the configuration for this command in Figure 5-47.

```

Address Phase
IF HWRITE = '1' THEN
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"6000" =>
    data_zero_en <= '1';
  WHEN X"6004" =>
    data_one_en <= '1';
  ...
  WHEN X"6010" =>
    data_four_en <= '1';

Data Phase
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"6000" =>
    data_zero <= HWDATA;
  WHEN X"6004" =>
    data_one <= HWDATA;
  ...
  WHEN X"6010" =>
    data_four <= HWDATA;

```

Figure 5-46 Snippet VHDL code for TIM Control unit for Write Actuator Data Command

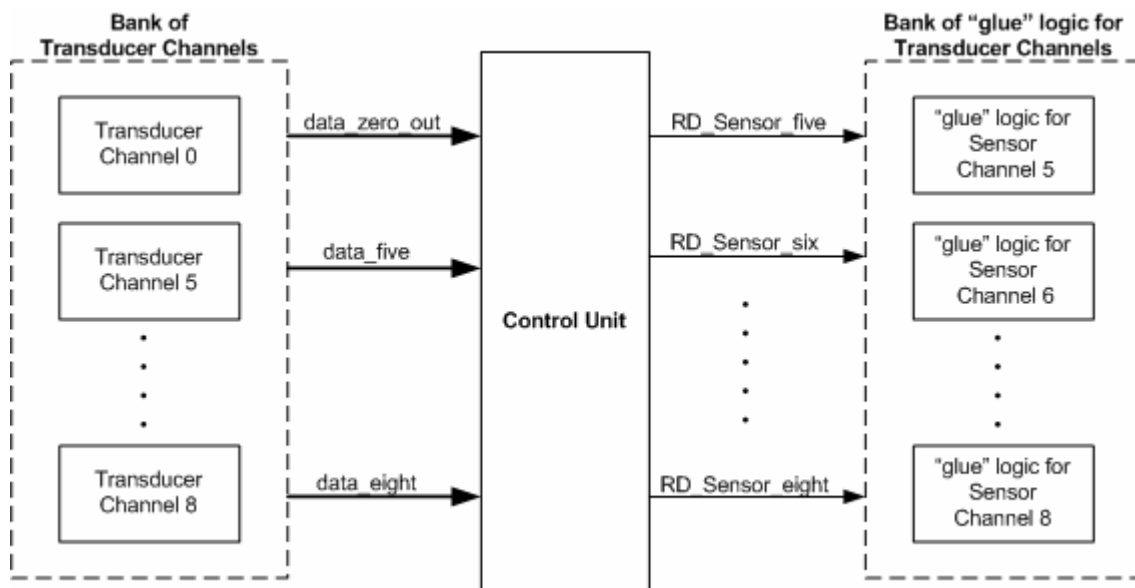


Figure 5-47 Control Unit Example Configuration for Read Sensor Data Command

Using this configuration, we can design the VHDL code so that during the address phase we set the control signals (*RD_Sensor_five* through *RD_Sensor_eight* shown in Figure 5-47), then during the data phase we place the sensor data register outputs (*data_zero_out*, *data_five*, etc.) onto the HRDATA signal of the bus. A snippet of the VHDL code that is needed for this command under the 8-Channel TIM configuration is shown in Figure 5-48.

```

Address Phase
IF HWRITE = '0' THEN
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"6014" =>
    RD_Sensor_five <= '1';
  WHEN X"6018" =>
    RD_Sensor_six <= '1';
  ...
  WHEN X"6020" =>
    RD_Sensor_eight <= '1';

Data Phase
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"6000" =>
    HRDATA <= data_zero_out;
  WHEN X"6014" =>
    HRDATA <= data_five_out;
  ...
  WHEN X"6020" =>
    HRDATA <= data_eight_out;

```

Figure 5-48 Snippet VHDL code for TIM Control unit for Read Sensor Data Command

Next, we design the configuration for the *Reset* command. This command places the TIM's registers in their default power-on state. Therefore, we need to generate a signal that resets the registers of the implemented transducer channels. This configuration is designed in Figure 5-49.

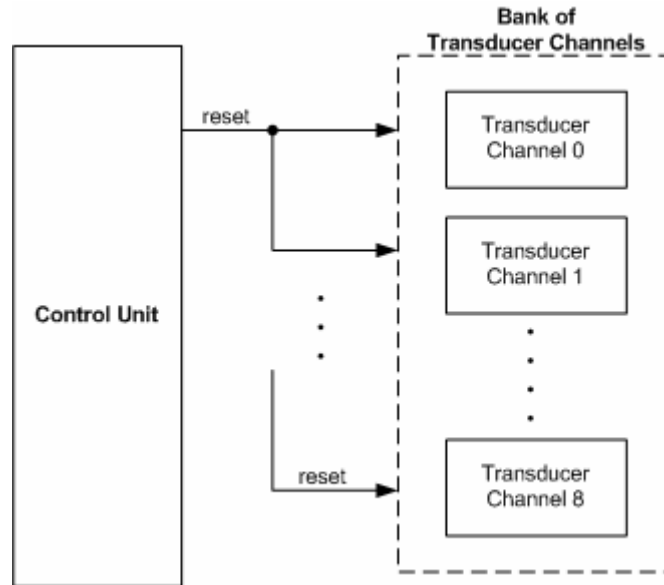


Figure 5-49 Control Unit Example Configuration for Reset Command

This command is asserted regardless of it is a read or write request on the bus. Therefore, when this command is selected, we set the reset signal during the address phase of the bus protocol. A snippet of the VHDL code that is needed for this command is designed in Figure 5-50.

```

Address Phase
CASE HADDRESS (15 DOWNT0 0) IS
  WHEN X"8000" =>
    reset <= '1';

```

Figure 5-50 Snippet VHDL code for TIM Control unit for Reset Command

Next, we will show the design of the transducer channels and their respective “glue” logic.

5.2.3 Transducer Channel Block

From the specifications chapter, we know that each individual transducer channel has three registers associated with it. These registers include a data register (for the sensor reading or

actuator data), a status register and an interrupt mask register. These registers are designed to be 32 bits since that is the width size of the bus. Also, each individual channel generates an interrupt signal from an AND operation between its status and interrupt mask register (there is a one-to-one relationship between the two). In order to represent this structure and allow for a high level of configurability, we design a generic transducer channel block that can be configured for a sensor or actuator.

The difference in the configurations is in the data registers. This is because for actuators, the control unit handles the control/data signals since the input data to this data register is set by the NCAP through the *Write Channel Data command*. Then, the output of this register is sent to “glue” logic so that it can set the physical actuator with this data when the NCAP sends a trigger command. On the other hand, when the channel is a sensor we need “glue” logic that interfaces with the sensor (through an ADC or digital inputs). This “glue” logic writes the sensor reading to the channel’s data register when the trigger command is asserted by the NCAP. Then, the output of the sensor data register needs to be sent to the control unit so that the NCAP can access this information through the *Read Transducer Channel Data* command. With this in mind, we design a generic transducer channel block in Figure 5-51.

To use the generic transducer channel block that is shown in Figure 5-51, we need to design “glue” logic that handles the status registers and the interface with the physical transducers.

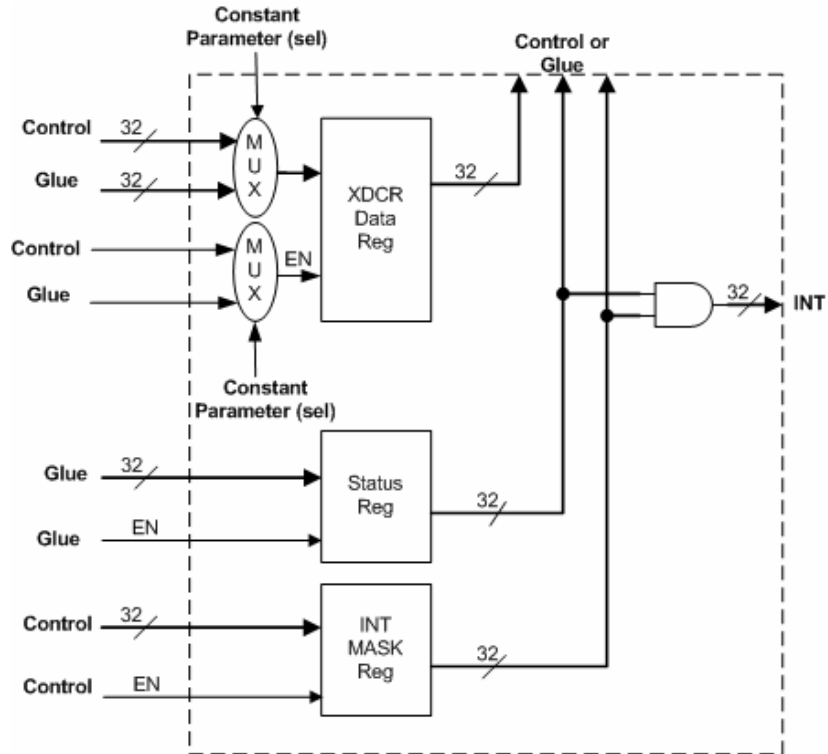


Figure 5-51 TIM Transducer Channel

Next, we will show the design for a sensor channel that is interfaced with an 8-bit ADC. Note that for the purpose of this example we are setting the ADC to continuously convert. For this configuration, we need to send the output of the registers to the TIM control unit so that the NCAP can access them. Also, we need the control unit to generate the interrupt mask register's control/data signals, as well as interface signals (*RD_Sensor* and *Trigger*) that are connected to the “glue” logic. The *RD_Sensor* signal indicates that the sensor is being read so the Trigger ACK bit of the status register should be cleared, while the *Trigger* signal is used to denote that the sensor should be read. In this case, the sensor information is read from the output of the 8-bit ADC. Since we set the ADC to continuously convert, we use the INT signal from the ADC to denote when the enable signal of the channel's data register should be set. With this information, we can design the top-level structure for a sensor channel that is interfaced with an 8-bit ADC in Figure 5-52.

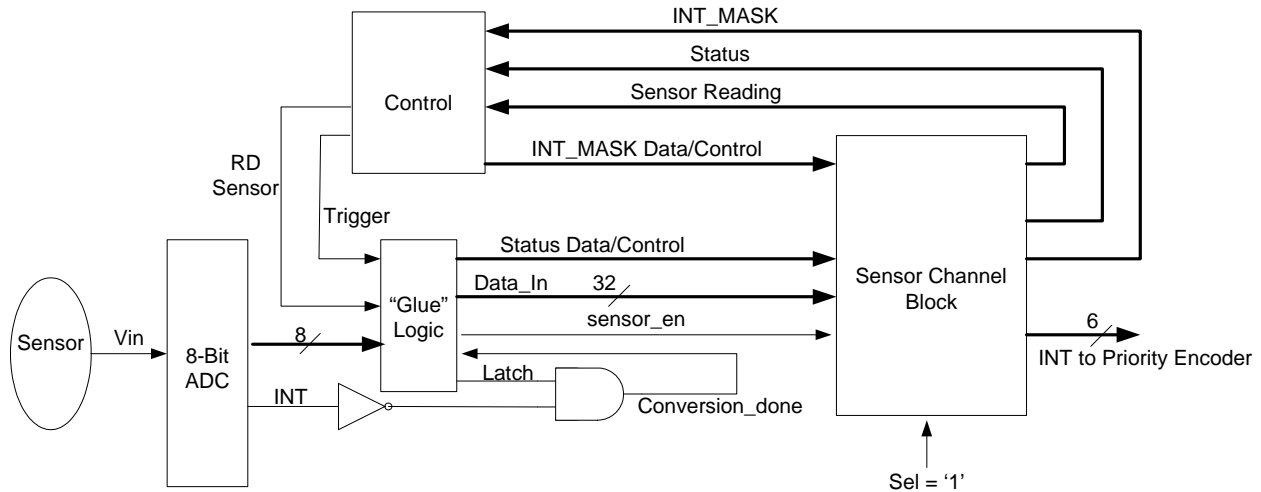


Figure 5-52 Sensor Channel Interface with an 8-bit ADC Configuration

Using this top-level structure, we can design the “glue” logic for a sensor channel under this configuration (8-bit ADC). The “glue” logic is responsible for generating the status register’s control/data signals, reading the sensor and setting the control/data signals of the channel’s data register. In order to do this, the “glue” logic has four inputs (*RD_Sensor*, *Trigger*, 8-bit output from the ADC, and *conversion_done*) and five outputs (Status Register enable and input data signals, sensor data register’s enable and input data and a *Latch* signal). Therefore, using this inputs/outputs, we can design three basic blocks (for the “glue” logic), a block that generates the control/data signals, a trigger control block, and a block that pads the output of the ADC into the 32 bits of the channel’s data register. The top-level structure for the “glue” logic that is needed for this sensor configuration is designed in Figure 5-53.

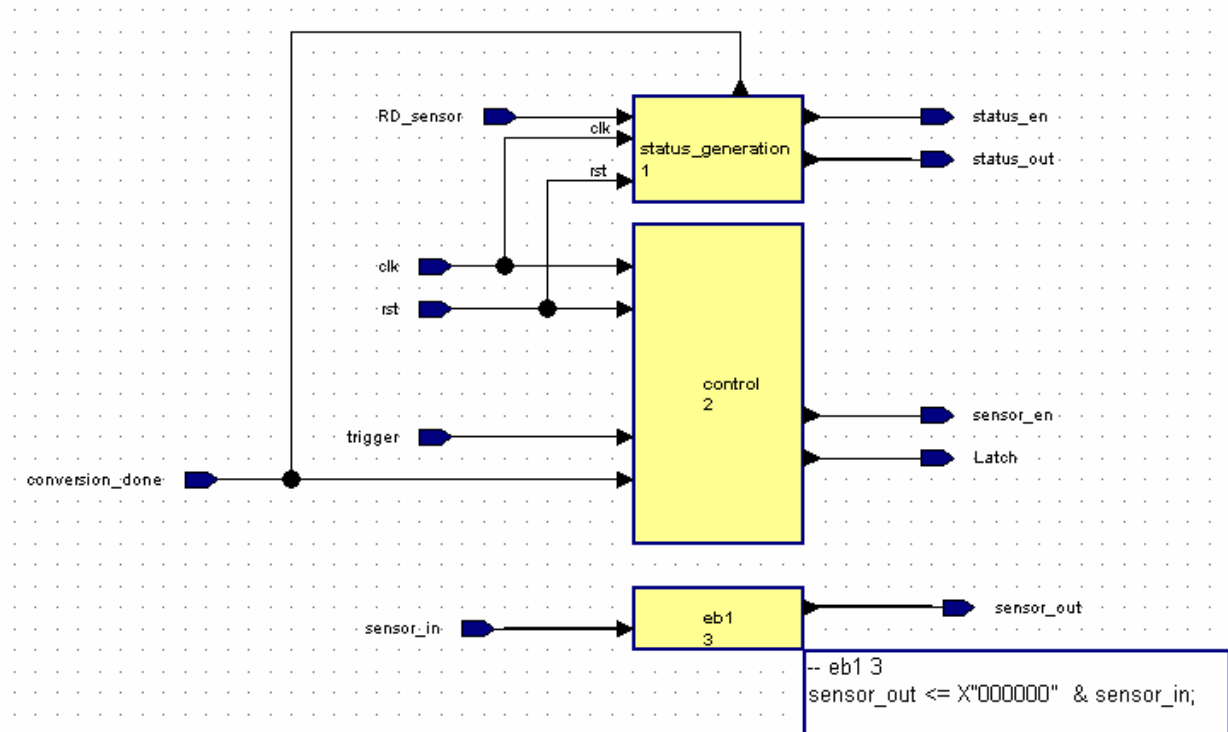


Figure 5-53 “Glue” Logic for Sensor Channel interfaced with ADC

The *status_generation* block (shown in Figure 5-53) sets the control/data signals of the status register. In order to do this, we follow the specifications that were derived in Section 4.2.4. The status bits are summarized in Table 5-4.

Table 5-4 Status Register Bits

Bit	TIM CHANNEL_ZERO	Individual Channel
0	CHANNEL_ZERO Trigger Acknowledge	Channel Trigger Acknowledge
1	Invalid Command	Reserved
2	TIM Operational	Channel Operational
3	Corrections enabled/disabled	Corrections enabled/disabled
4-31	For future use	For future use

The *Trigger ACK* bit is set when the *conversion_done* signal is asserted. Then, we clear this bit when the *RD_sensor* (generated by the control unit when the NCAP issues a *Read Transducer Data command*) signal is asserted.

The *Channel Operational* bit is asserted upon power-up, since the sensor channel is able to respond to commands after it has been powered-up.

The *Corrections enabled/disabled* bit is always set to zero, since the TIM does not have capabilities to apply corrections to the sensor data (e.g. convert temperature into Celsius).

In order to generate the bits described above, we design VHDL code in Figure 5-54.

```
process (clk, rst, RD_sensor)
begin
    if rst = '0' then
        status_en <= '0';
        status_out <= (others => '0');
    elsif (rising_edge(clk)) then
        status_en <= '1';
        -- Trigger ACK
        if conversion_done = '1' then
            status_out <= "00000000000000000000000000000101";
            -- Clear trigger ACK
        elsif RD_sensor = '1' then
            status_out <= "00000000000000000000000000000100";
        else
            status_out <= "00000000000000000000000000000100";
        end if;
    end if;
end if;
end process;
```

Figure 5-54 VHDL Code for Status Generation Block for a sensor channel that is interfaced with an 8-bit ADC

Next, we design the control that is used to handle the trigger behavior of the sensor channel. When we receive a trigger command (trigger signal is asserted), we need to set the *Latch* signal for the duration until the *conversion_done* signal is a '1'. Then, when this occurs we need to set the enable signal of the sensor data register. Therefore, in order to design this block we use a state machine with three states. An idle state in which we are waiting for a trigger command, then when the sensor channel is selected for a trigger command we move on to the converting state. In this state we assert the *Latch* signal (previously shown in Figure 5-52 until the ADC finishes conversion, which happens when the *conversion_done* (output of AND operation between the *Latch* and *INT* signals previously shown in Figure 5-52) signal is '1'. Then, when this occurs we know that the NCAP has finished conversion so we enable the sensor

data register and set the Latch signal to zero. The design of this state machine is shown in Figure 5-55.

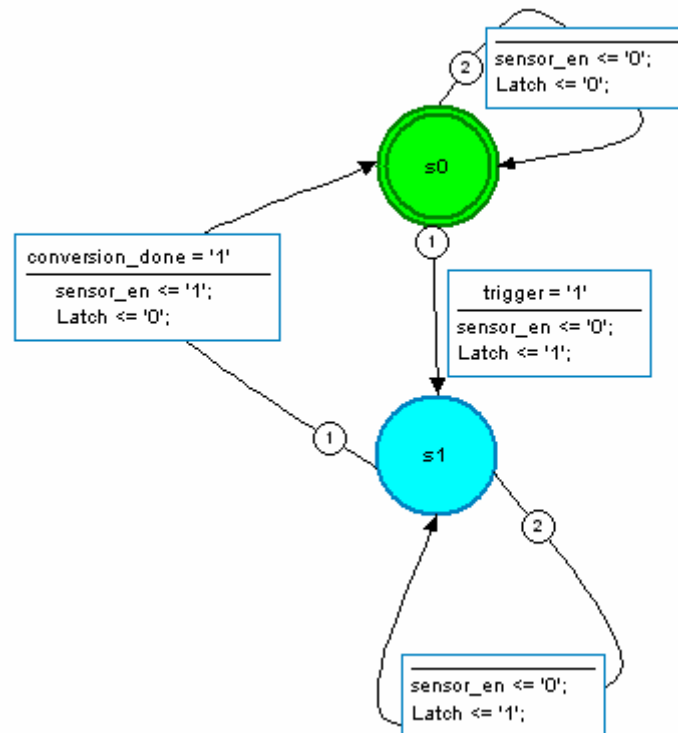


Figure 5-55 “Glue” logic State Machine for Sensor Channel interfaced with ADC

Note that the output of the ADC is padded with zeroes using combinational logic (as was shown in Figure 5-53 by the *eb1* block) so that this value can be sent to the data register.

Next, we show the configuration for a sensor channel that is interfaced with digital inputs. The different connections for this configuration are similar to what was previously shown for the sensor channel that is interfaced with the 8-bit ADC. The only difference is that the sensor output is directly connected to the “glue” logic instead of using an ADC or external logic for the interface. With this in mind, we design the configuration for a sensor channel that is interfaced with digital inputs in Figure 5-56.

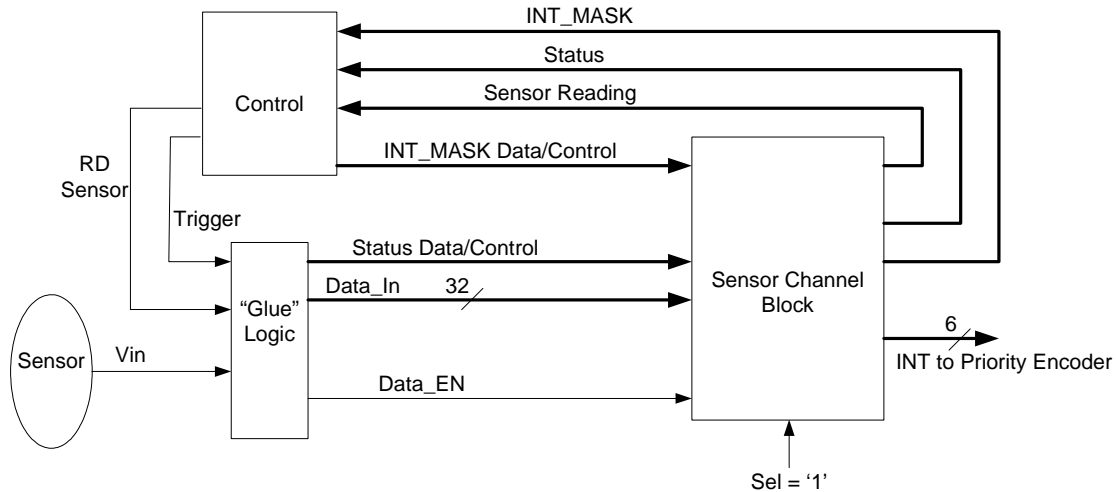


Figure 5-56 Sensor Channel Interface with digital I/O Configuration

Next, we can show the design for the “glue” logic portion of this sensor channel configuration. Note that the structure for the “glue” logic is also similar to what was done for the 8-bit ADC sensor channel configuration. The only difference is that the trigger ACK status bit can be generated upon reception of the trigger since there is no conversion to be made and only a latch of the sensor reading has to be made. With this in mind, we design the top-level structure for the “glue” logic in Figure 5-57.

The *status_generation* block remains unchanged from the example that was shown before. Therefore, to generate the control/data bits for the status register, we can use the VHDL code that was previously designed in Figure 5-54. The only block that requires any changes is the control block. This is because as was mentioned earlier, we do not need to wait for the ADC to send an *INT* signal (as was done in the ADC sensor channel example) since we are using digital inputs. Therefore, the state machine for the “glue” logic only needs two states, an idle state (waiting for trigger command) and an execute state in which we set the *sensor_en* enable signal of the sensor data register so that the *sensor_out* signal can be written to the sensor register. Also, during the execute state we set the *done* signal that is interfaced with the

status_generation block. This is done so that the Trigger ACK bit of the channel's status register can be generated in the same way as was done for the 8-bit ADC sensor channel example. The state machine for the “glue” logic is designed in Figure 5-58.

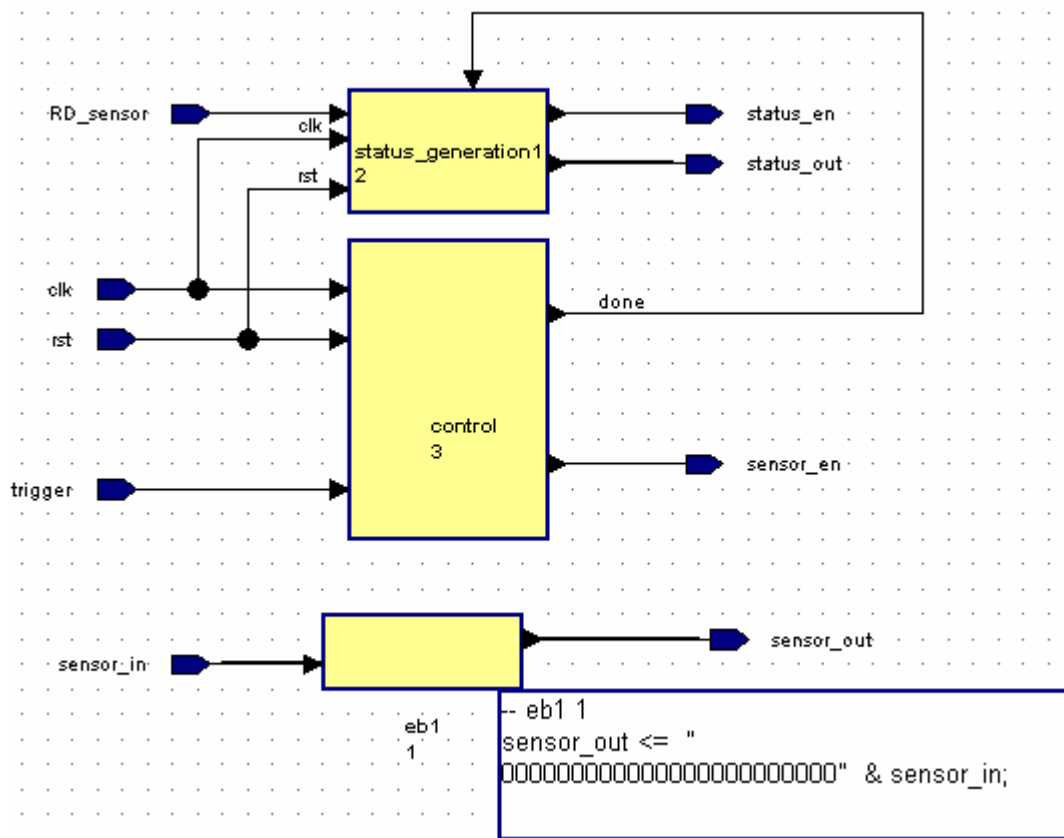


Figure 5-57 “Glue” Logic for Sensor Channel interfaced with digital I/O

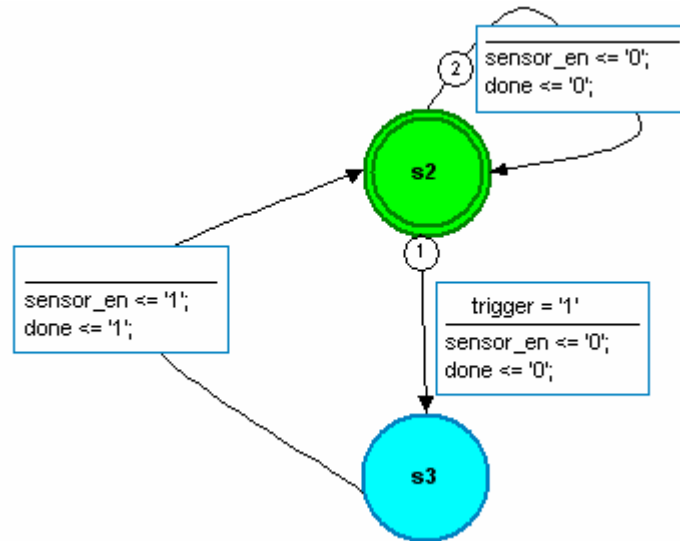


Figure 5-58 “Glue” logic State Machine for Sensor Channel interfaced with digital I/O

Next, we design an example configuration for an actuator channel that is interfaced with an 8-bit DAC. This configuration is different than what has been shown for the sensor channels since the actuator channel data register is written by the NCAP (by means of the *Write Transducer Channel Data* command). Therefore, the control unit sets the control/data signals of the channel’s data and interrupt mask registers. Then, the output of the status and interrupt mask registers is sent to the control unit so that it can be accessed by the NCAP (using the *Read Status Register* and *Read Interrupt Mask Register* commands). Also, the “glue” logic sets the control/data signals of the status register and it interfaces with the control with the actuator channel block (receives the output of the actuator channel data register), the control unit, and the 8-bit DAC (sets the actuator with the data set that is stored in the data register upon reception of a trigger command). With this in mind, we design the configuration for an actuator channel that is interfaced with an 8-bit DAC in Figure 5-59.

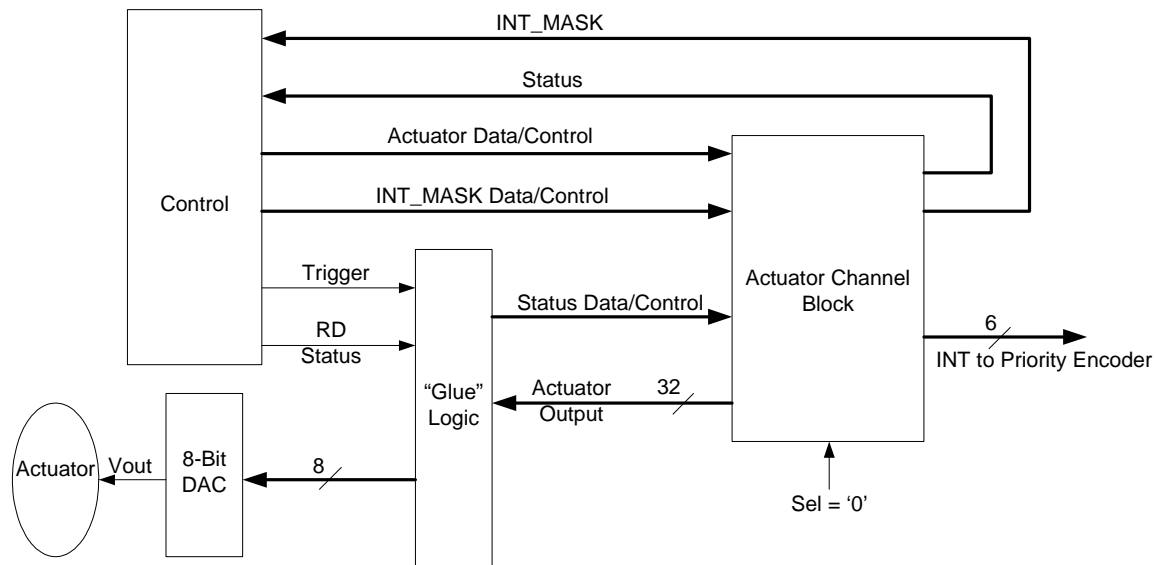


Figure 5-59 Actuator Channel Interface with an 8-bit DAC Configuration

Next, we need to design the “glue” logic for this configuration. In order to do this we can still use three main blocks in a similar fashion to what was done for the “glue” logic that was designed for the sensor channels. Therefore, we need a block that generates the control/data signals of the status register, we also need a control block (used to handle triggering), and a combinational logic block that grabs the least significant eight bits of the actuator channel’s data register (for the interface with the 8-bit DAC). Note that these eight bits need to go through a register before we send them to the DAC. This is because the NCAP writes the data set to the transducer data register before sending the trigger command. Then, the physical actuator is set with this data when a trigger command is received. With this in mind, we design the “glue” logic for this actuator channel configuration in Figure 5-60.

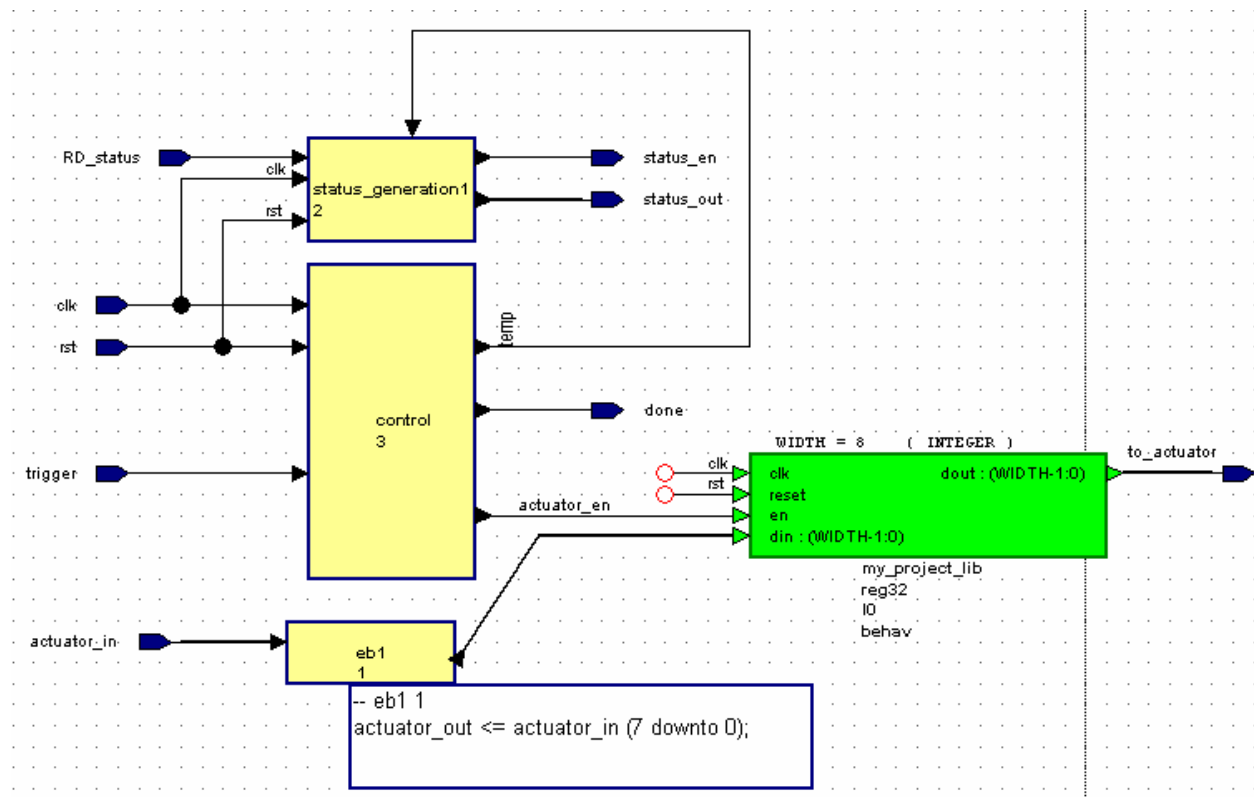


Figure 5-60 “Glue” Logic Architecture for an Actuator Channel interfaced with DAC

Next, we show the design of the *status_generation* block. This block needs to use the RD_Status (generated by the control unit when the NCAP issues a *Read Transducer Data command*) that is used to clear the Trigger ACK status bit. For the two other status bits (*Channel Operational* and *Conversions enabled/disabled*) we use the same scheme as what was done for the sensor channels. Therefore, the *Channel Operational* bit is always one, while the *Conversions enabled/disabled* bit is always zero. The only thing that this block must do then is to generate the Trigger ACK status bit. This bit is set when we the actuator acquires the data set (“glue” logic register is enabled), and when this happens the temp signal (shown in Figure 5-60 as an input to the status_generation block) is asserted. Therefore, using this signal we set the Trigger ACK bit of the status register, and we clear this bit when the RD_Status signal is asserted. To do this, we can use the VHDL that was previously designed for the sensor channel

configuration in Figure 5-54. However, we would have to replace the *RD_Sensor* signal with the *RD_Status* signal, but the logic remains the same.

Next, we design the block that handles the trigger behavior of the actuator. In order to do this, we use a state machine in a similar fashion to what has been done in the “glue” logic that has been designed for the sensor channels. This state machine has to generate three output signals (*done*, *temp*, and *actuator_en*). The *done* and *temp* signals are asserted when the actuator has been successfully set (trigger ACK), while the *actuator_en* signal is asserted upon reception of the trigger command (set the actuator). To do this, we need two states (much like what was previously designed in Figure 5-58) that are designed in Figure 5-61.

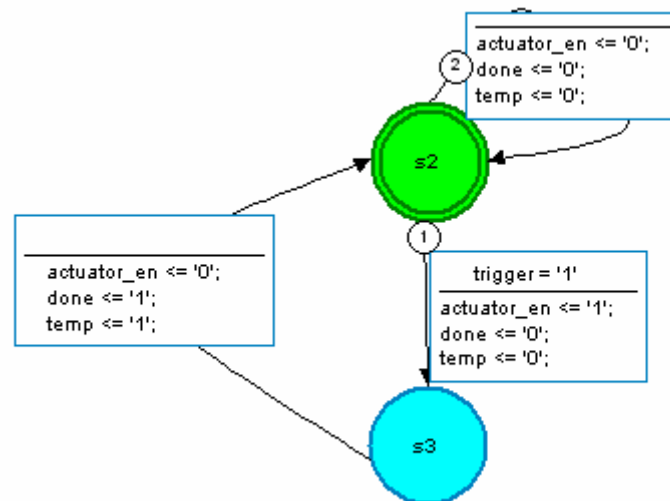


Figure 5-61 “Glue” logic State Machine for Actuator Channel interfaced with DAC

Next, we show the design for an actuator channel that is interfaced with digital I/O. The configuration for this actuator channel is similar to that of an actuator channel that is interfaced with a DAC. The only difference is in the combinational logic block (*eb1* from Figure 5-59) since it depends on the number of digital outputs that are used for the actuator. For example, if we use three digital outputs, then we send the least significant three bits of the actuator channel’s

data register to the “glue” logic’s register. With this in mind, we design the configuration for an actuator channel that is interfaced with digital I/O in Figure 5-62.

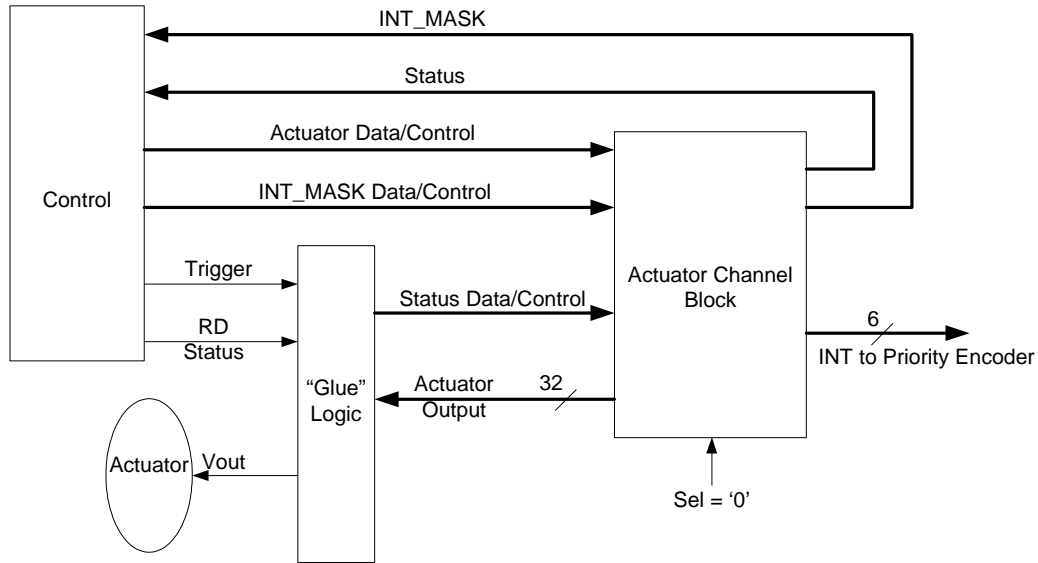


Figure 5-62 Actuator Channel Interface with digital I/O Configuration

As was mentioned earlier, the design of the “glue” logic does not change much, since we only need to change the *ebi* block that was designed in Figure 5-60 to use the actuator data that is needed to set the physical actuator. Consequently, we also need to change the width size of the “glue” logic’s register, which is simple because it is re-configurable. Therefore, we only need to change the WIDTH parameter that was shown in the register representation in Figure 5-60.

The transducer channel block that we have designed in this section is easy to reuse for individual channels that are implemented within the TIM. However, it is important to note that the CHANNEL_ZERO block is different since it is dependent on the number of transducer channels that are implemented. Therefore, this block needs to represent the actuator data as well as the sensor readings of all implemented channels. To do this, we design the structure shown in Figure 5-63.

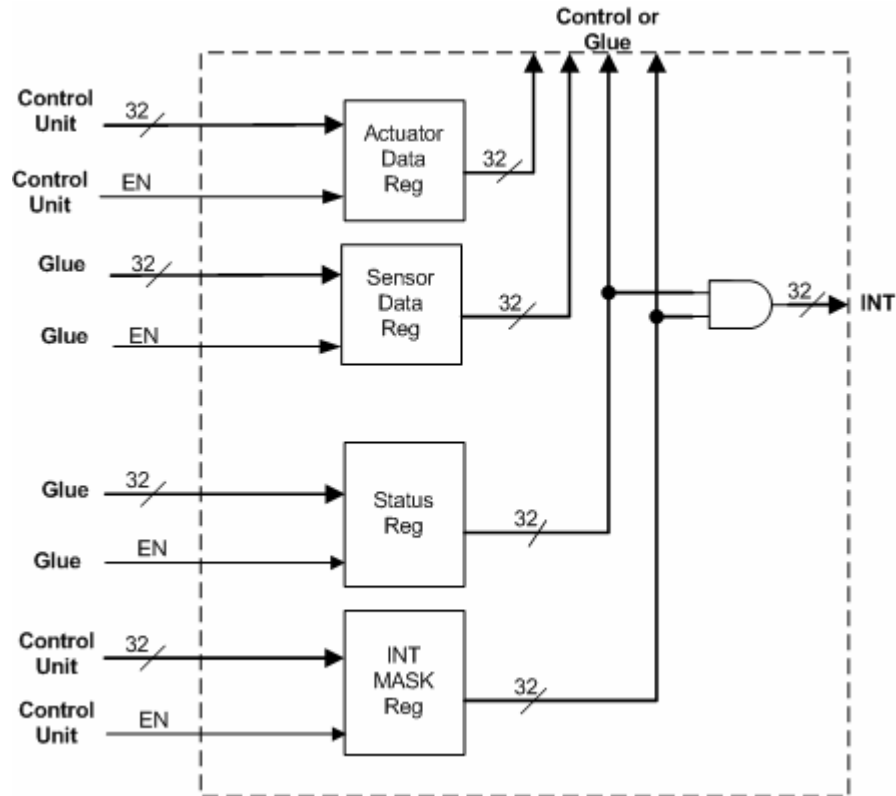


Figure 5-63 TIM Transducer CHANNEL_ZERO Structure

Note that this structure has two sets of data registers, one for the actuators and one for the sensors. The control unit controls the actuator register, while the “glue” logic does the same for the sensor register. We use these new registers because, as was mentioned earlier, CHANNEL_ZERO needs to represent the result of all the sensor readings and all the actuator registers. Therefore, by having two extra registers, the control unit’s interface to the TIM channels does not change. Rather, we use the “glue” logic for CHANNEL_ZERO to manipulate the transducer data.

As an example of this logic, for an 8-Channel TIM implementation, we would use a similar configuration to what has been shown for the individual channels. However, the “glue” logic for CHANNEL_ZERO interfaces with the “glue” logic of the individual channels as well as the control unit. As far as the interface with the individual channel’s “glue” logic, we need to

receive the Trigger ACK signals from the individual channels. This is because when a global trigger is sent, CHANNEL_ZERO Trigger ACK is generated only after all individual channels have generated their respective Trigger ACKs. For the control unit's interface, we need to receive an *Invalid* (generated by the control unit to signal that the NCAP tried to issue a command that is invalid or not implemented) and *RD_Status* signals. Both of these signals are used to generate the status register control/data signals. Also we need to send the outputs of CHANNEL_ZERO's sensor data, interrupt mask, and status registers to the control unit so that they can be accessed by the NCAP. In order to handle the various cases that we have described, we can design the configuration for CHANNEL_ZERO for an 8-Channel TIM implementation in Figure 5-64.

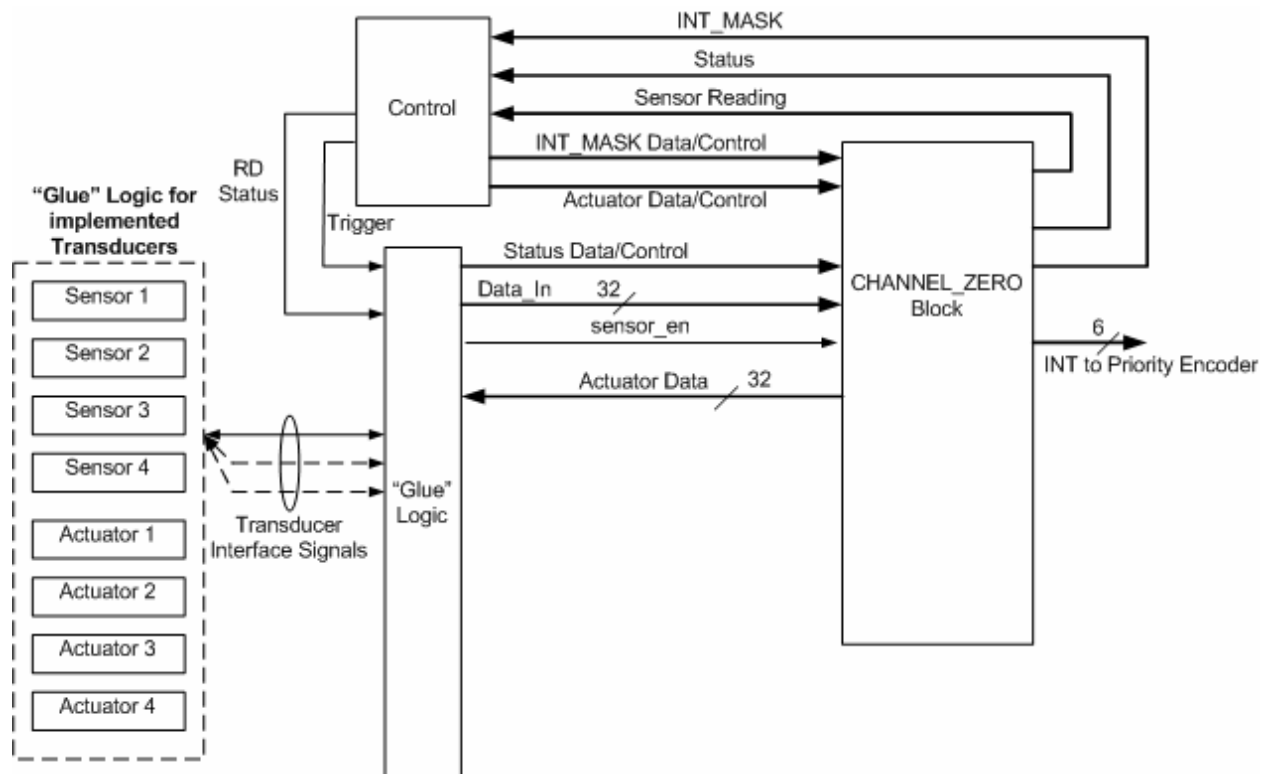


Figure 5-64 CHANNEL_ZERO Example for 8-Channel TIM Configuration

Again, the structure for the “glue” logic for CHANNEL_ZERO is similar to the “glue” logic of the individual channels. So, we use three main blocks (status_generation, trigger control, and combinational logic for the transducer data). The design of this top-level structure is shown in Figure 5-65.

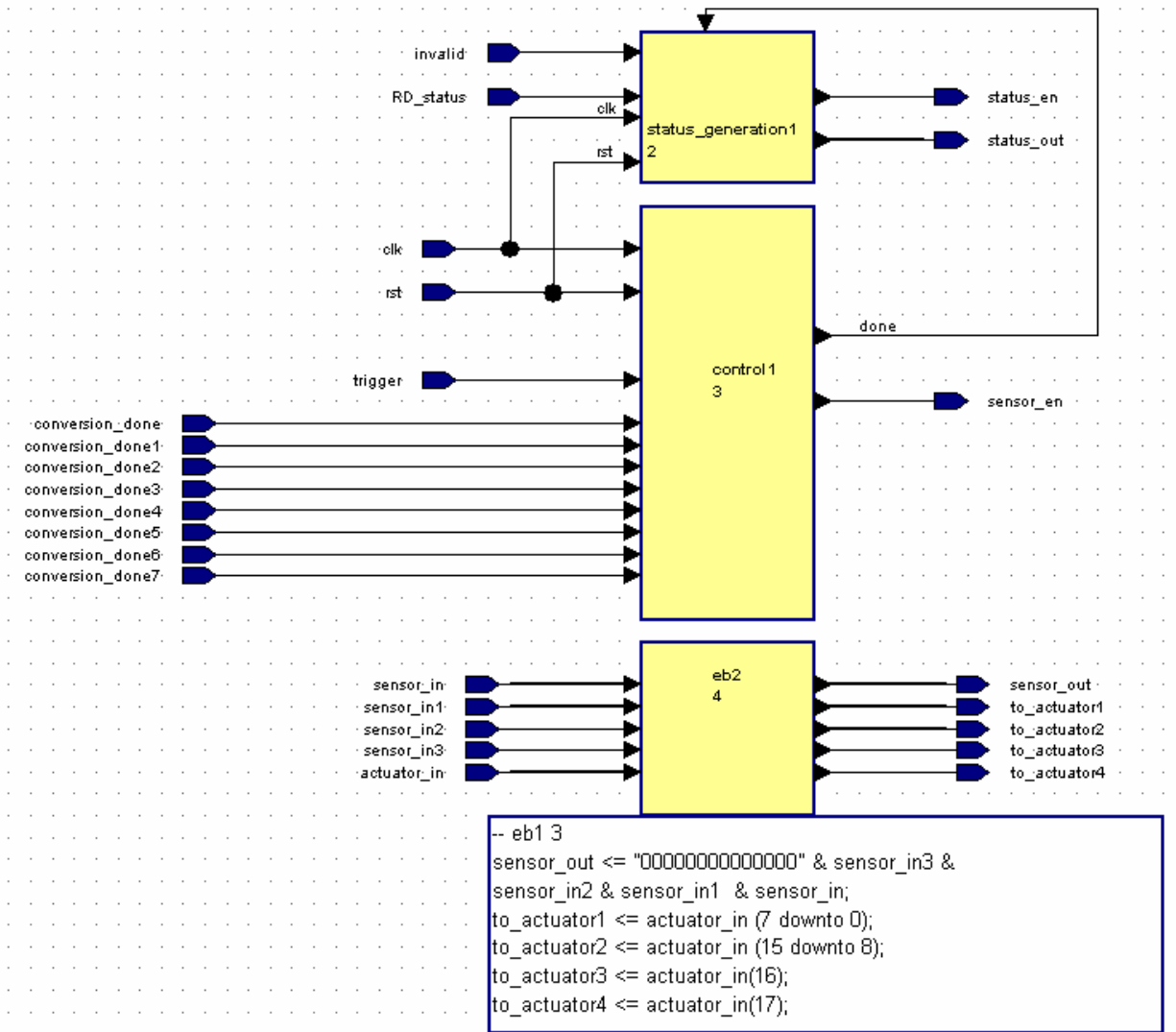


Figure 5-65 “Glue” Logic for CHANNEL_ZERO

The *status_generation* block uses information from the “glue” logic’s controller as well as the TIM’s control unit to set the different status bits. The *Trigger ACK* bit is asserted when every individual channel has been successfully sampled/set and this bit is cleared when the TIM

control unit asserts the *RD_Status* signal. Also, this block asserts the *invalid command* bit when the control unit asserts the *Invalid* signal (shown in Figure 5-64) and this bit is cleared when the *RD_Status* signal is asserted. The two other status bits (TIM operational and Corrections enabled/disabled previously shown in Table 5-4) are set to one and zero upon power-up. This is because the TIM can respond to NCAP commands upon power-up (TIM Operational) and it does not have capabilities to apply corrections to the transducer data.

Next, we can design the control for the “glue” logic. As was designed for the individual channels, we use a state machine to control triggering behavior of the channel. For CHANNEL_ZERO, this is more complicated since when a global trigger is applied, every implemented channel is triggered. Therefore, CHANNEL_ZERO sends the output of its actuator channel’s data register to the “glue” logic of the individual actuator channels. On the other hand, we enable the sensor data register when the individual sensors have been successfully sampled so that we can write the sensor readings onto CHANNEL_ZERO’s sensor data register.

So, the state machine that we are designing needs to set the *done* (denotes the Trigger ACK condition) and *sensor_en* signals when every implemented channel has executed its individual trigger. To set these signals, we must wait until every *conversion_done* (trigger ACK for individual channels) signal is asserted. With this in mind we can design the control for the “glue” logic of CHANNEL_ZERO for this 8-Channel TIM example in Figure 5-66.

In this section, we have shown the complete design for the representation of transducer channels (both individual and CHANNEL_ZERO) within a TIM. For more information on the VHDL code for the Transducer Channel Blocks and their corresponding “glue” logic, refer to Appendices C and D.

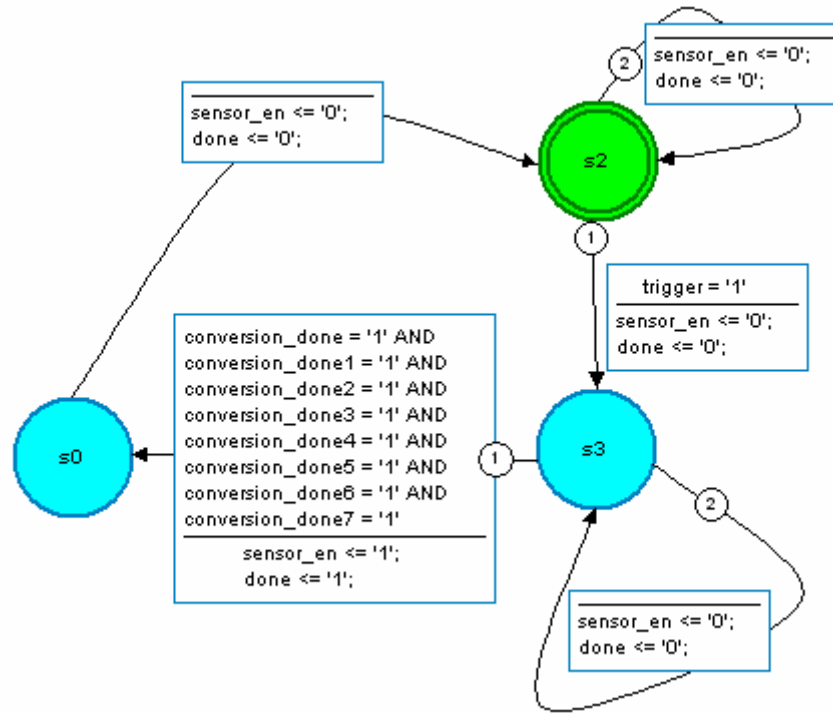


Figure 5-66 State Machine for “Glue” logic’s control

5.2.4 Interrupt Management

The interrupt signal for each channel is generated from the transducer channel block that was discussed in the previous section. In order to handle these interrupts, we need an interrupt controller. The Excalibur chip provides an interrupt controller that can be used. This interrupt controller generates two interrupt signals (FIQ and IRQ) to the embedded processor, and it can handle up to 63 individual interrupt requests coming from the PLD. The structure for this interrupt controller is shown in Figure 5-67.

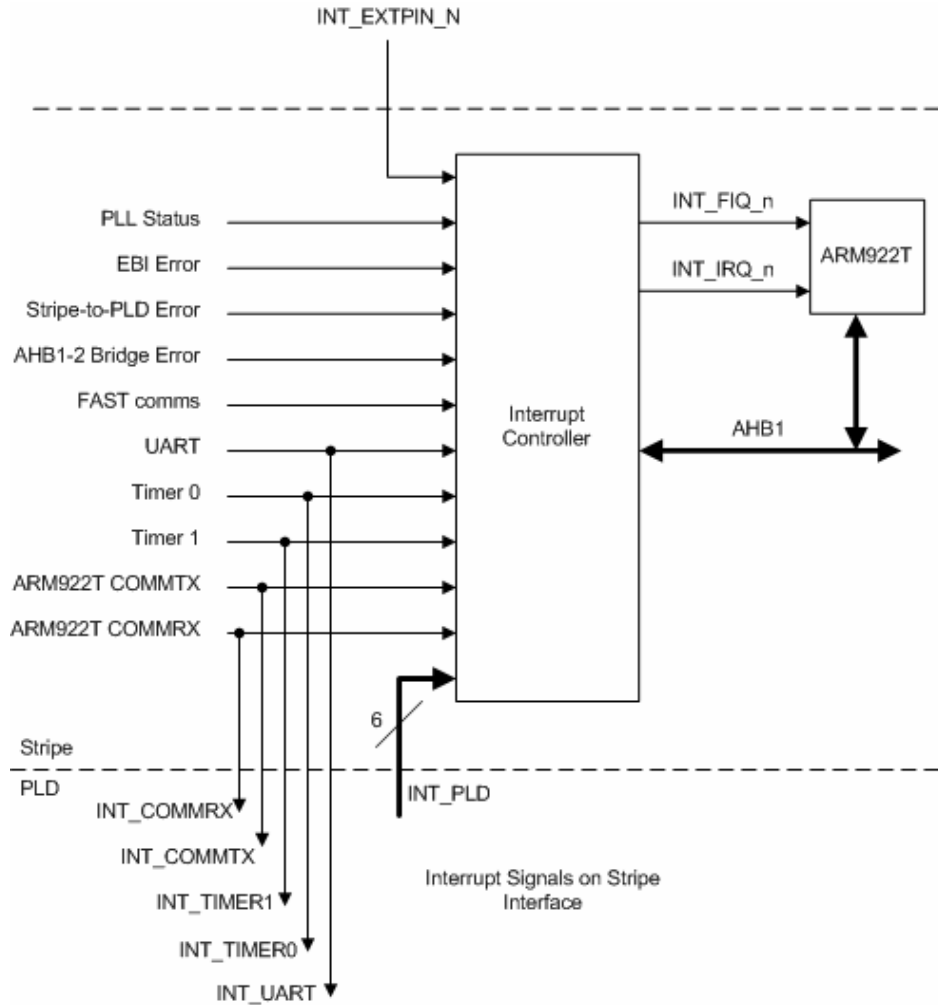


Figure 5-67 Interrupt Controller overall structure

From the figure, we can see that there are 17 different interrupt sources. Out of these sources inputs, we have 10 interrupts within the stripe (PLL Status, EBI Error, Stripe-to-PLD Error, AHB1-2 Bridge Error, Fast comms, UART, Timer 0, Timer 1, ARM922T COMMTX, ARM922T COMMRX), one external pin (INT_EXTPIN_N), and six from the PLD stripe interface (where the TIM resides) as an interrupt bus (INT_PLD). These six bits can be configured in three different modes. These modes are as follows:

- Each PLD interrupt signal is interpreted as an individual interrupt. This is the default mode.

- The signals are interpreted as a single interrupt request, using a six-bit priority value.
- The signals are interpreted as a single interrupt request, using a five-bit priority value together with one individual interrupt

Using this interrupt controller with the 6-bit priority mode, a maximum of 19 implemented transducer channels can be handled. This is assuming that only the mandatory status bits are implemented. Therefore, we would have four possible interrupts from CHANNEL_ZERO, and three possible interrupts from each individual channel.

However, if the application requires that more interrupts be handled, either because more status bits, more channels are implemented, or a bigger chip is used. Designers can implement their own interrupt controller within the PLD. This is made possible by the Excalibur chip's architecture, since the outputs from the other interrupt sources within the stripe (such as UART or timer), are made available as inputs to the PLD as was shown in Figure 5-67.

Next, we design the functionality of the interrupt controller. This block has a priority scheme that allows the embedded processor to determine who generated the interrupt. To use this priority scheme, we write a unique priority value (between 1H and 3FH) to priority registers (provided by the Excalibur chip within the embedded stripe) during system initialization. Then, the prioritization logic within the interrupt controller constantly compares the priority values within all the priority registers to determine which interrupts are pending, and provides the value of the highest in the interrupt identification register INT_ID. With this information, the interrupt service routine reads this register to identify the interrupt source. Therefore, since the interrupt controller already has a built-in priority scheme, all we have to design is a priority encoder (as

was previously shown in Figure 5-32) that encodes the generated interrupts from the transducer channels into six bits (since we are using the 6-bit priority mode).

The interrupt priority logic of this block is as follows. The LSB of CHANNEL_ZERO has the highest priority and the MSB of channel 19 has the lowest priority. It is important to note that even though our design architecture allows for a maximum of 255 transducers, the priority block we design only handles up to 19 transducers. This is because as was mentioned earlier by using the interrupt controller that is provided this is the maximum amount of transducers that we can handle. Also, since the Excalibur chip that we have available (EPXA1) is the smallest version of the chip, we would not likely be implementing very large transducer systems because of the resource constraints that were discussed in Section 5.2. However, in the case that the application requires more interrupts, then it is the designer's responsibility to create a priority encoder to handle these changes. For more information on how to implement/handle more interrupts, refer to Appendix D. Next, we show the software routines that are used to handle the generated interrupts.

5.2.4.1 Interrupt Service Routine

The interrupts are handled in software by different handlers that are specific to the different interrupts. In order to initialize the interrupt controller and all the interrupts, we create a function *irq_init()*. This function initializes the interrupt controller to the six-bit priority mode, sets the priorities and the values of all the registers (TIM interrupt mask, UART, timer, etc). Note that if the designer implements a new interrupt controller, then he/she would have to edit this initialization routine.

Then, when the processor receives an IRQ interrupt, there is another function *ClrqHandler()* that is called. This function checks the ID register to see who generated this

interrupt. Then, using case statements we determine which interrupt handler to use for this interrupt. Because of the fact that the interrupt controller can handle a maximum of 61 TIM interrupts (4 from CHANNEL_ZERO and 3 for each individual channel), we set up the *CirqHandler()* function to jump to 61 different handlers for the TIM. However, since these handlers depend on the application, it is the programmer's responsibility to implement their functionality. It is important to note that the software that is written to handle the TIM interrupts is part of the TIM firmware (previously shown in Figure 5-1). Therefore, it is transducer-specific code that may vary from application to application.

5.2.5 Summary and Constraints

The architecture that we have described for the TIM provides users with a great deal of configurability, which can be exploited for their particular application. This is because to add a transducer, users only have to “drop in” a transducer channel block and then implement its particular “glue” logic. However, as was mentioned earlier in the specifications chapter, the number of transducers that can be interfaced to the TIM is limited by the resources of the Excalibur chip. The following discussion goes into detail about the various sizes of the Excalibur chip and the maximum amount of transducers that can be interfaced to the TIM.

As was previously stated in Section 4.2, the I/O of the Excalibur chip ranges from 186 in the EPXA1 to 711 in the EPXA10. This I/O count is the total available user I/O (shared stripe I/O + PLD I/O). However, in the EPXA1 all the stripe I/O is dedicated so we only have 40 user-defined PLD I/O pins that can be customized for an application. This is not the case for the EPXA10 in which only the SDRAM Clock Enable pin is dedicated on the Stripe, and all other pins stripe pins are shared with the PLD. Therefore, using the EPXA10 designers have a

maximum of 708 I/O pins that they can use (note that around 100 of these pins must be used for AHB communications). Next, we show the maximum amount of transducers that can be implemented with each version of the chip.

The maximum number of transducers that can be implemented in the EPXA1 chip is 40. Note that to do this, we would have to use a single digital I/O bit for each transducer. So, if the application requires that the transducers interface to data converters or more than a single I/O bit, the maximum number of transducers that can be implemented decreases. For example, if we use 8-bit data converters (assuming two bits for control) for each implemented transducer, we can implement a maximum of four transducers. This is because we would need 10 bits for each ADC/DAC (eight for data and two for control). Depending on the application, a designer can use a variety of combinations to take advantage of the 40 I/O pins that are provided by the EPXA1 chip.

On the other hand, if the EPXA10 chip is used, then the designer could conceivably implement the maximum amount of transducers that are allowed by the standard (255). To implement this number of transducers, designers would have to use a combination of data converters and digital I/O bits to interface to every transducer.

6.0 IMPLEMENTATION AND TESTS

The implementation was done using Quartus II from Altera, as well as external circuitry for the transducers. This tool was used to compile the hardware and software that was designed. The system was tested by using hyperterminal as the debugging tool, having the software output relevant information about the system's behavior to the terminal by the use of an RS-232 connection.

6.1 APPLICATION SYSTEM

In order to test our NCAP/TIM combination, we built a proof of concept application system that consisted of a three channel TIM. The transducers in this test system consisted of two sensors (temperature and light), and LEDs to simulate an actuator. The object of this system was not to build a precise instrumentation system, but rather it was to verify the design met the specs and, hence, the standard. Next, we will describe the configurations for the sensors and how they were interfaced to the TIM.

For the temperature sensor, a thermistor was used. This sensor is a component that shows a large change in resistance with a change in its body temperature. There are two types of thermistors, large positive temperature coefficient of resistance (PTC devices) and large negative

temperature coefficient of resistance (NTC devices). The thermistor that was used was Jameco's 102-NTC⁽²⁷⁾.

To read the thermistor, a voltage divider was used. This is because the thermistor's resistance changes with respect to the temperature. This way, there is a fixed resistance in parallel with the thermistor and the temperature is obtained by reading the voltage through an ADC. Then, a table can be created with the different voltage values that the different temperatures would yield according to the device's datasheet.

According to the device's datasheet the resistance when at 25 Celsius varies 4.3% every time the temperature changes by a degree. Because of the nature of NTC thermistors, this means that as the temperature goes down the resistance increases and as the temperature goes up the resistance decreases.

Using a simple voltage divider with a 5 Volt source and 1200 ohm resistor as is depicted in the Figure 6-1, we can calculate a table with information of the temperatures with respect to the voltages (V_t) so that a look-up table can be created with this values and the conversion of raw voltage data to physical units can be simplified.

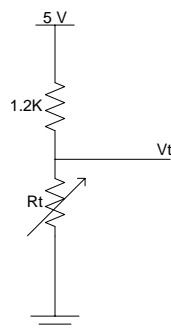


Figure 6-1 Thermistor's Connections

Now we can calculate the voltage with respect to the temperature using the thermistor's data sheet (reference). This is shown in Figure 6-2. Note that two plots are shown in the figure.

The first one shows the thermistor's resistance vs. the temperature in Celsius, while the second plot shows the thermistor's voltage vs. the temperature in Celsius. The thermistor's voltage was then interfaced to an ADC so that the readings can be interfaced to the TIM. The ADC that was interfaced with the thermistor was the ADC0820 8-bit ADC converter from National Semiconductor ^(28,29). This ADC uses a single supply power voltage of 5 V max, the reference voltages are completely differential, and it has a conversion time of 2.5 μ s. With this information, we can begin to discuss the ADC's interface with TIM portion of the Excalibur chip. The ADC's control and output signals were interfaced to "glue" logic that resided within the PLD. In order to meet the Excalibur chip's electrical specifications, we powered the ADC with a 5V power supply and the ADC's signals were interfaced directly with the TIM.

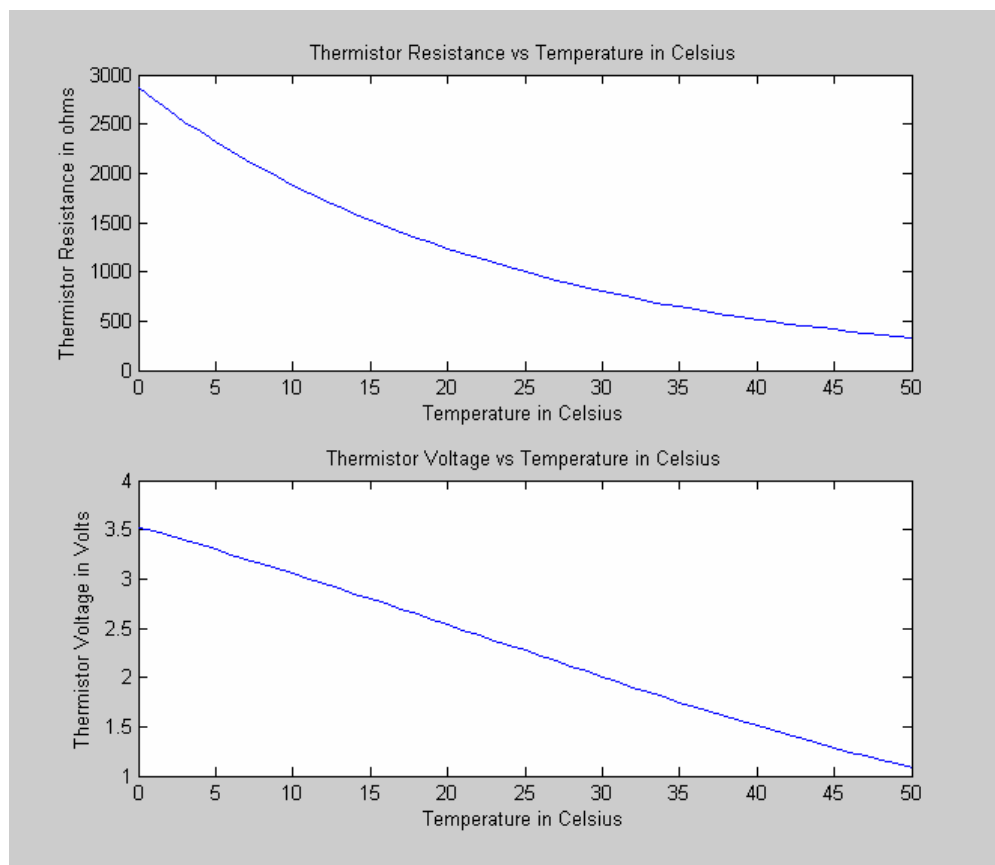


Figure 6-2 Thermistor's behavior plots

The thermistor's range with this configuration goes from 0 to 50 Celsius with voltages from 1V to 3.6V. Therefore, in order to fully utilize the range that the ADC provides, we configured the voltage references (since they are completely differential) to use 1V as its V_{ref-} and 3.6V as its V_{ref+} by using a voltage divider as is shown in the Figure 6-3.

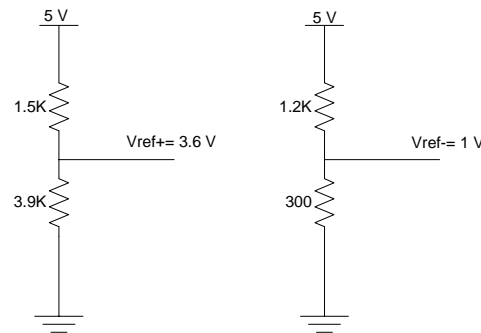


Figure 6-3 ADC Voltage dividers

Note that in here we did not take into account the resistance of the ADC chip itself which is 1.5K ohms according to the datasheet. So, when the connections were made with the ADC, the chip's parallel resistance changed the specified voltages that are shown in the Figure 6-3. The new measured voltages were 2.9V for V_{ref+} and 1.2V for V_{ref-} , which gives a range of operation for the thermistor of 16 to 48 Celsius which is sufficient for this type of application.

With this information the ADC output was matched to the actual temperature in Celsius by doing a curve-fitting of the Voltage vs. Temperature plot shown in Figure 6-2. The results from this mapping are shown in Table 6-1.

It is important to note that an application-specific software operation was created to convert the raw readings of the sensor temperature into Celsius. This was done using the information shown in the table previously discussed. Also, note that the complete resolution (8 bits) of the ADC was used by configuring the differential reference voltages to the minimum and maximum temperatures that the thermistor could handle.

Table 6-1 Thermistor's ADC Output and Corresponding Temperature Value

HEX Value	Temperature in Celsius
FE	16
F6	17
ED	18
E5	19
DD	20
D5	21
CD	22
C5	23
BC	24
B4	25
AC	26
A4	27
9B	28
93	29
8B	30
82	31
7A	32
72	33
6A	34
62	35
5A	36
53	37
4B	38
44	39
3C	40
35	41
2E	42
27	43
20	44
19	45
13	46
C	47
6	48

The other sensor that was used is a photoconductive cell ⁽³⁰⁾ that is used to detect the amount of light in the environment. This photocell is similar to the thermistor in that it has a variant resistance with respect to the light intensity (Lumens), with the rate of change being 130 ohms for every 100 Lumens.

For the purpose of this project, the sensor is used to detect if there is light in the room, which simplifies the interface to the TIM. This way, the sensor's readings are directly interfaced to digital IO of the Excalibur chip. Figure 6-4 shows the connections for this sensor.

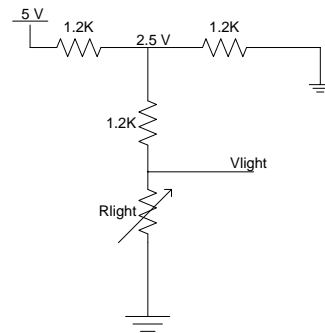


Figure 6-4 Photocell's connections

Note that the voltage that is used to “power” the photocell is only 2.5V. This is to obtain a range in which a change in light can be easily detected by digital IO of the Excalibur chip¹⁸. This is because the characteristics of the chip dictate that the voltage that it recognizes as a digital zero is less than 1V, and a digital one is considered to be greater than 1.65V.

The Excalibur chip also provides a variety of on-chip LEDs that were used to simulate an actuator. The LEDs are set up to signal if there is light or not according to the photocell and to represent the room temperature by three states low, normal, or high. These connections are shown in Figure 6-5 Note that the outputs go through a NOT gate since the LEDs are active low.

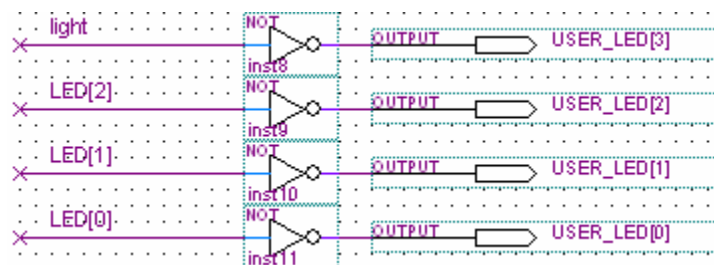


Figure 6-5 LED Connections

Next, we show a picture of the complete system implementation. This is shown in Figure 6-6. The external circuitry was interfaced to the Excalibur chip by means of a 40-pin header that was connected to J15 in the development kit. The IO was directly wired to the user-defined IO that was given to the designer through the APEX20K FPGA.

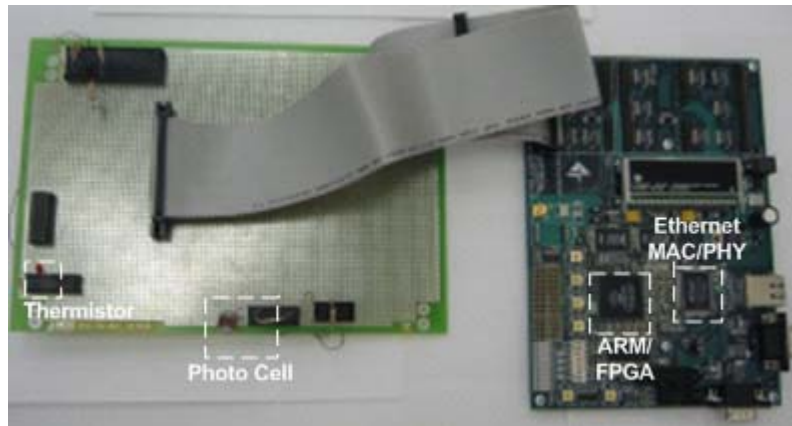


Figure 6-6 Picture of the Implementation

Next, we will show the proof of concept implementation of the complete system generation along with the NCAP and TIM implementations.

6.2 SYSTEM IMPLEMENTATION

The complete IEEE 1451 system was implemented based on the design that was previously presented. This implementation consisted of a combination of hardware and software modules. We can summarize the complete implementation of the single chip solution in Figure 6-7.

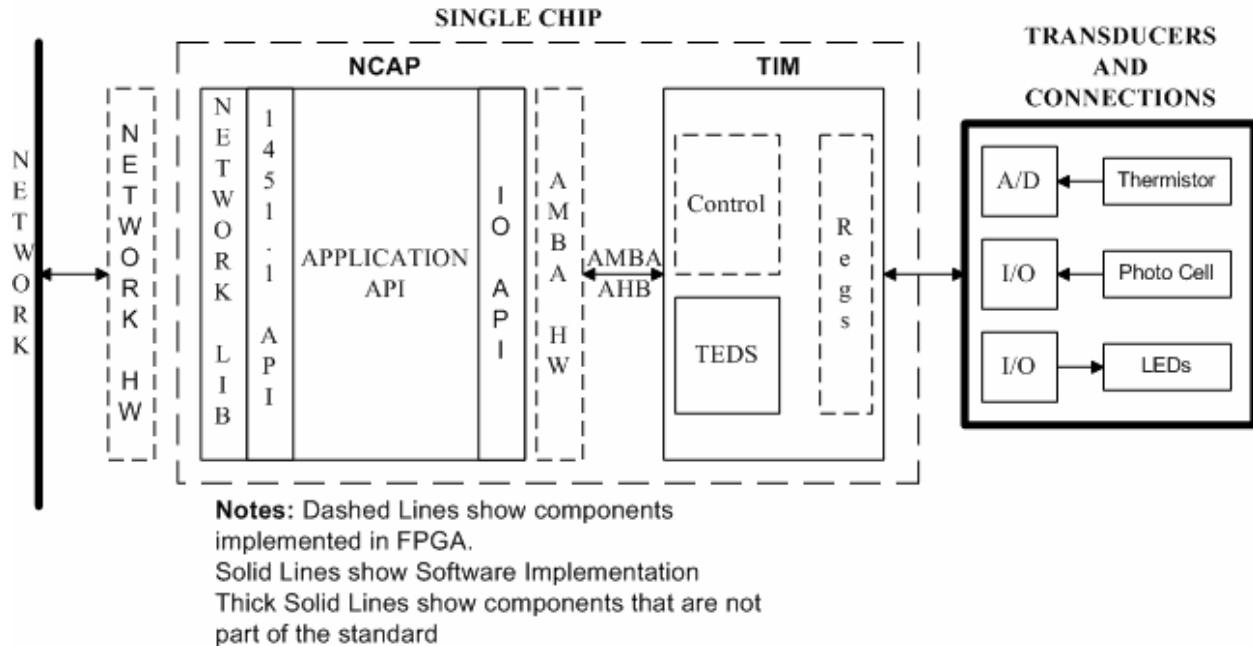


Figure 6-7 System Implementation

Next, we discuss the network hardware interface. For this interface, we compared the resource usage (of the hardware implementation) of three networks. These networks consisted of an Ethernet Network, CAN, and Universal Serial Bus (USB). All of these networks fit the scope of the standard.

The hardware implementations were compared using Open Core IP that was targeted for an Altera APEX20K FPGA (PLD present in the Excalibur chip). The results were used for a device utilization comparison. The results obtained were as follows:

- For an Ethernet MAC Controller designed by Cast, Inc. the device consumed 4,082 Logic Elements (LEs), 17 embedded array blocks (EABs) and 200 I/O pins in the FPGA ⁽²³⁾.
- For a CAN Controller designed by Bosch, the utilization was 4515 LEs, and 46 I/O pins in the FPGA ⁽²⁴⁾.

- For a USB Controller designed by Cast, Inc. the device utilization was 1,502 LEs, 12 Embedded System Blocks (ESBs), and 49 I/O pins ⁽²⁵⁾.

As was mentioned in both the Specifications and Design chapters, the EPXA1 chip provides a maximum of 40 user-definable I/O pins. Therefore, it is easy to see that for the network interfaces that are discussed above, the hardware for these networks cannot be implemented in the FPGA. However, it is important to note that if we had any of the bigger size Excalibur chips available, we could implement any of the three network implementations that were discussed.

Because of this resource constraint of the EPXA1 chip, we used an Ethernet MAC/PHY chip ⁽²⁶⁾ that was provided by the chip's development kit. This Ethernet MAC/PHY chip was interfaced to the External Bus Interface (EBI), so the NCAP had direct access to it through this peripheral. This was done to use the example configuration that was provided by the Ethernet MAC/PHY chip. Also, this chip provided a software library that was used for the network infrastructure.

This software library provided an operation, *smc_init()*, that initialized the Ethernet MAC/PHY chip. This operation is network-specific and it is called during the initialization phase of the NCAP in such a way that is transparent to the user. Other operations that were provided, dealt with the physical communications of the chip with the Ethernet network. For our implementation, we created an operation called *MarshalAndTransmitReceive()* that took as an input, an *ArgumentArray*, and returned an *OpReturnCode* (standard-defined return type) so that it could be easily interpreted by the network operations that are defined in the standard. The signature of this operation and a brief explanation is shown next.

```
OpReturnCode MarshalAndTransmitReceive(/* in*/ ArgumentArray in,  
/* in */int execute_publish)
```

This operation first checks the `execute_publish` value to see if the *Execute* or *Publish* operations need network access. This was done so that the different scenarios of the client/server and publish/subscribe models could be handled. Then, if it was the *Execute* operation, we marshaled the *ArgumentArray* onto the Ethernet network's format (packet of bytes) and did the loopback test with this information. Note that after the data was received we called the *Perform* operation, to follow the specifications that were shown in Section 4.1.2.4. On the other hand, for the *Publish* operation, we only sent and received the message to/from the network infrastructure.

By defining this operation in this way, we assured that it “fit” the network communication API of the standard, since we used all the types and behaviors of the *Execute* and *Publish* operations (network-side API of the 1451.1 standard).

6.2.1 Network Capable Application Processor Implementation

To implement the physical structure of the NCAP we used Altera's Mega plug-in wizard. Using this wizard we configured the complete embedded stripe, which included the ARM as well as the signals that allow the processor to communicate with the PLD and others peripherals. This configuration was made using the design decisions shown in Chapter 5.0. Next, we show this configuration.

First, we used the Boot from Flash option so that we would permanently store the program memory in the flash memory that was provided by the Excalibur chip. Then, because the TIM is a slave on the embedded stripe as was discussed in the design chapter, we used the

Stripe-to-PLD interface (PLD configured as a slave). Also, to handle every possible generated interrupt we configured the interrupt controller for the 6-bit priority mode.

The next step was to configure the bus clock speed. The bus was configured to run at 160 MHz for AHB1, and 80 MHz for AHB2. This was done so that the performance of the NCAP/TIM communication would be faster than the Transducer Independent Interface (TII) of the 1451.2 standard.

We set every memory range/address space for the peripherals that are part of the embedded stripe. Note that we selected a memory range of 64K for the PLD (TIM) so that every possible implemented channel and command could be represented as was mentioned in Section 5.2.2. After completing all these steps, the complete embedded stripe block is generated. This generated block is shown in Figure 6-8.

As far as the software is concerned, we followed the design example shown in Section 5.1.5. Therefore, we instantiated an NCAP Block class (to control the experiment) and a Transducer Block class (to map the TIM's transducers). The individual transducers were represented as parameters and were instantiated after the TEDS was read in a similar fashion to what was shown in Section 5.1.5. On the other hand, the network ports were mapped during compile-time. These ports consisted of an Entity, Client Port, Publisher Port and Subscriber Port object classes. Next, we show the configuration for the network operations that were used in this application. We begin with the *Execute* operation, which is the client-side construct for client/server communications. For this operation we used the pseudo-code that was previously shown in Figure 5-7, so the only thing that we had to change was the network-specific call to the network infrastructure, which was done through the *MarshalAndTransmitReceive()* operation that was previously described in this section. The code for this operation is shown in Figure 6-9.

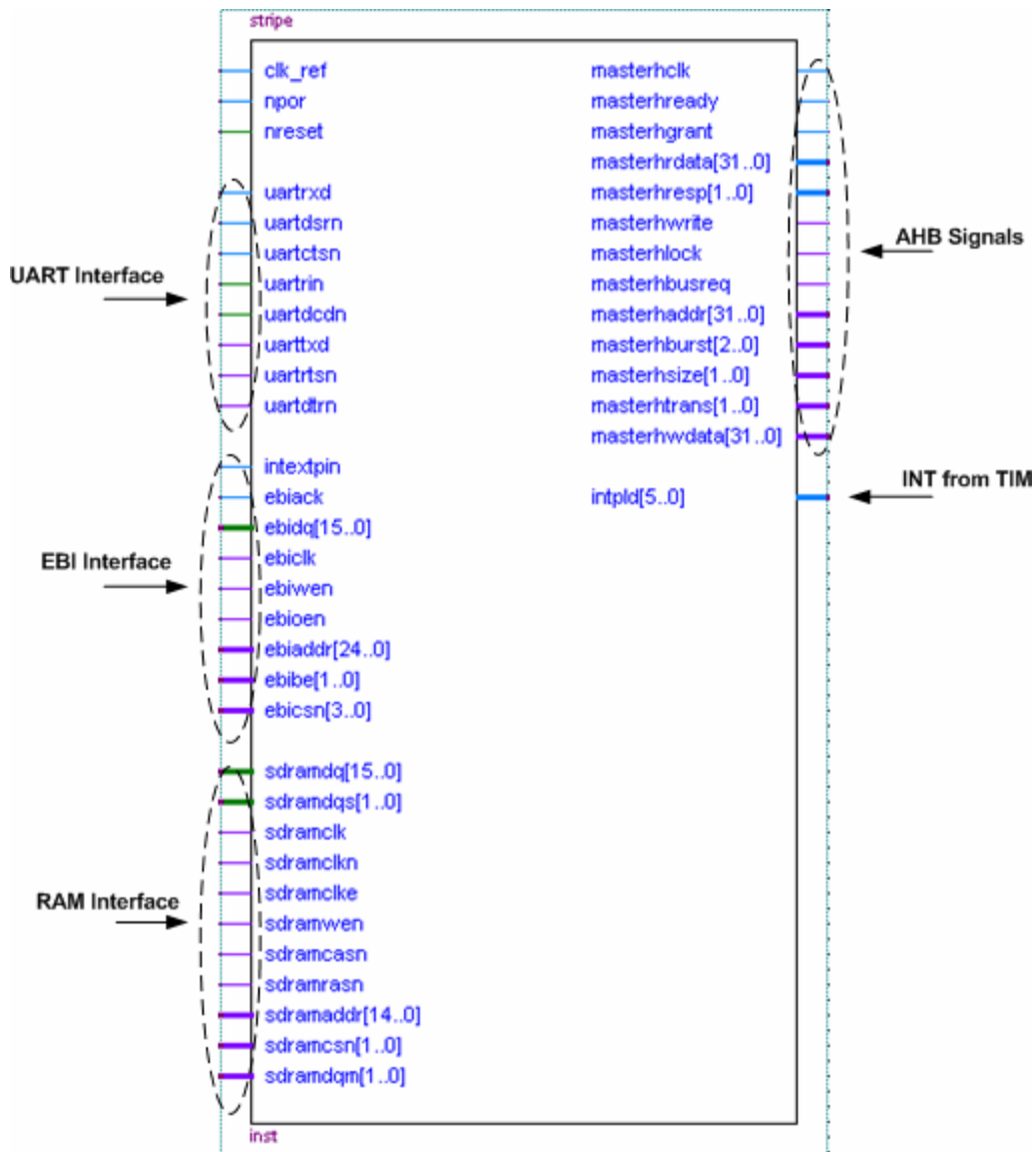


Figure 6-8 System's Embedded Stripe

Finally, we show the configuration for the Publish operation. This operation was the publisher-side construct for publish/subscribe operations. Because this operation only consisted of a call to the network infrastructure, we simply called the *MarshalAndTrasmitReceive()* operation in the code. The complete code for these network operations is shown in Appendix B.


```

if (execute_mode == 0)
{
    // Next line changes depending on the network infrastructure
    MarshalAndTransmitReceive(server_input_arguments, 0);
    // Timeout routine
    While (interrupt == 0)
    {
        // Make sure that the operation does not timeout
        If (jiffies > timeout)
        // Operation Timed out so return ClientServerReturnCode 14
        Return 14;
        Else
        // Operation Completed successfully
        Return 0;
    }
}
else
{
    // Next line changes depending on the network infrastructure
    MarshalAndTransmitReceive(server_input_arguments, 0);
    Return 0;
}

```

Figure 6-9 Execute() C Code

6.2.2 Transducer Interface Module Implementation

The Transducer Interface Module (TIM) that was implemented consisted of three channels. Channel 1 was a temperature sensor and was interfaced to an 8-bit ADC. Channel 2 was a light sensor that was interfaced to a digital I/O bit within the Excalibur chip, and channel 3 consisted of LEDs that were used to simulate an actuator. The TIM's structure was implemented using the design structure previously shown in Figure 5-32. Because of the high-level of configurability of the blocks that we designed in Chapter 5.0, a designer only needs to instantiate these blocks and then make minor changes to the VHDL code depending on his/her application. For example, if the designer is interfacing to a 12-bit ADC instead of an 8-bit ADC, then in the combination block of the "glue" logic (eb1 block previously shown in Figure 5-53), we set the sensor_out signal to use the least significant 12 bits (instead of eight bits as was shown in the example) and

pad the most significant bits with zeroes. Next, we show the different blocks that were instantiated for this proof of concept implementation in Figure 6-10.

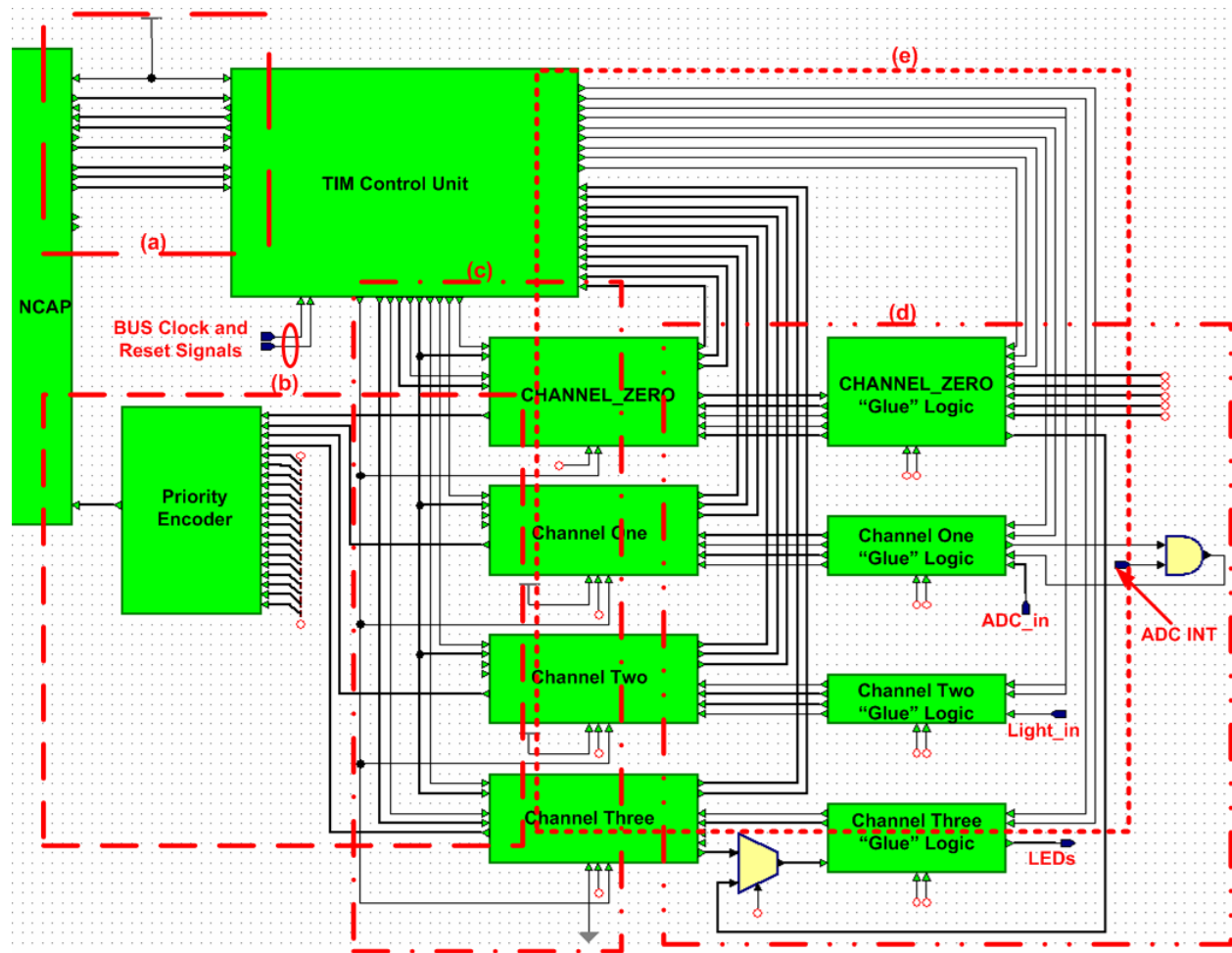


Figure 6-10 Top Level System Implementation

We instantiated four Transducer Channel Blocks (for CHANNEL_ZERO, the temperature sensor, the light sensor and the LEDs) along with their “glue” logic (both blocks were designed in Section 5.2.3), then these blocks were also interfaced with the Control Unit (used for NCAP communication) and with the priority encoder (used for the interrupts). Next, we describe the different connections that were made between the Control Unit and the NCAP (part b) of Figure 6-10). This is shown in Figure 6-11.

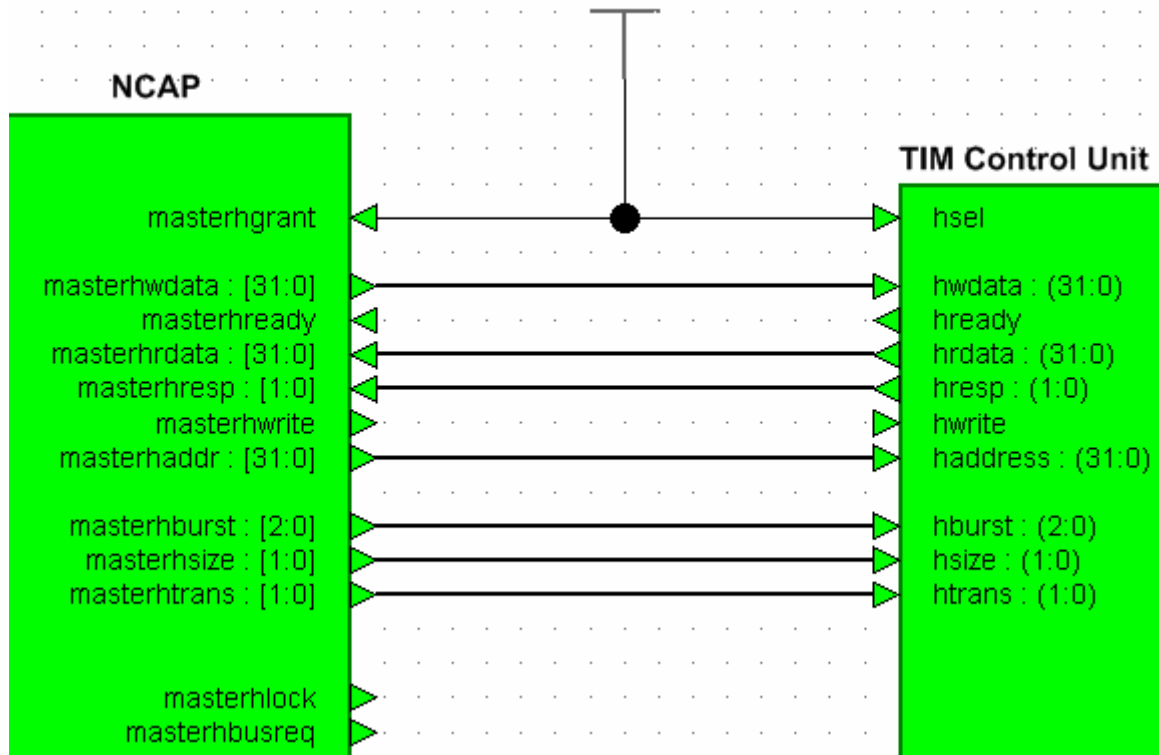


Figure 6-11 NCAP/TIM Connections, part (a) from Figure 6-10

Note that we connected the MASTER_HGRANT and HSEL signal to power. This was done for simplicity since we only had one slave within the PLD, so the NCAP is always granted access to the bus and the HSEL signal is always asserted. However, this does not mean that the TIM is always selected since from the VHDL code (shown in Section 5.2.2 and Appendix D), we use the HTRANS signal to see when the bus is active. Also, the MASTER_HLOCK and MASTER_BUSREQ signals are unused so they do not need to be connected. This is because the NCAP does not need to lock the bus for any type of transaction, nor do we have to send the request signal to an arbiter, since the MASTER_GRANT signal is always asserted. Next, we discuss the connections (part b in Figure 6-10) between the NCAP, the Priority Encoder and the Transducer Channels. This is shown in Figure 6-12.

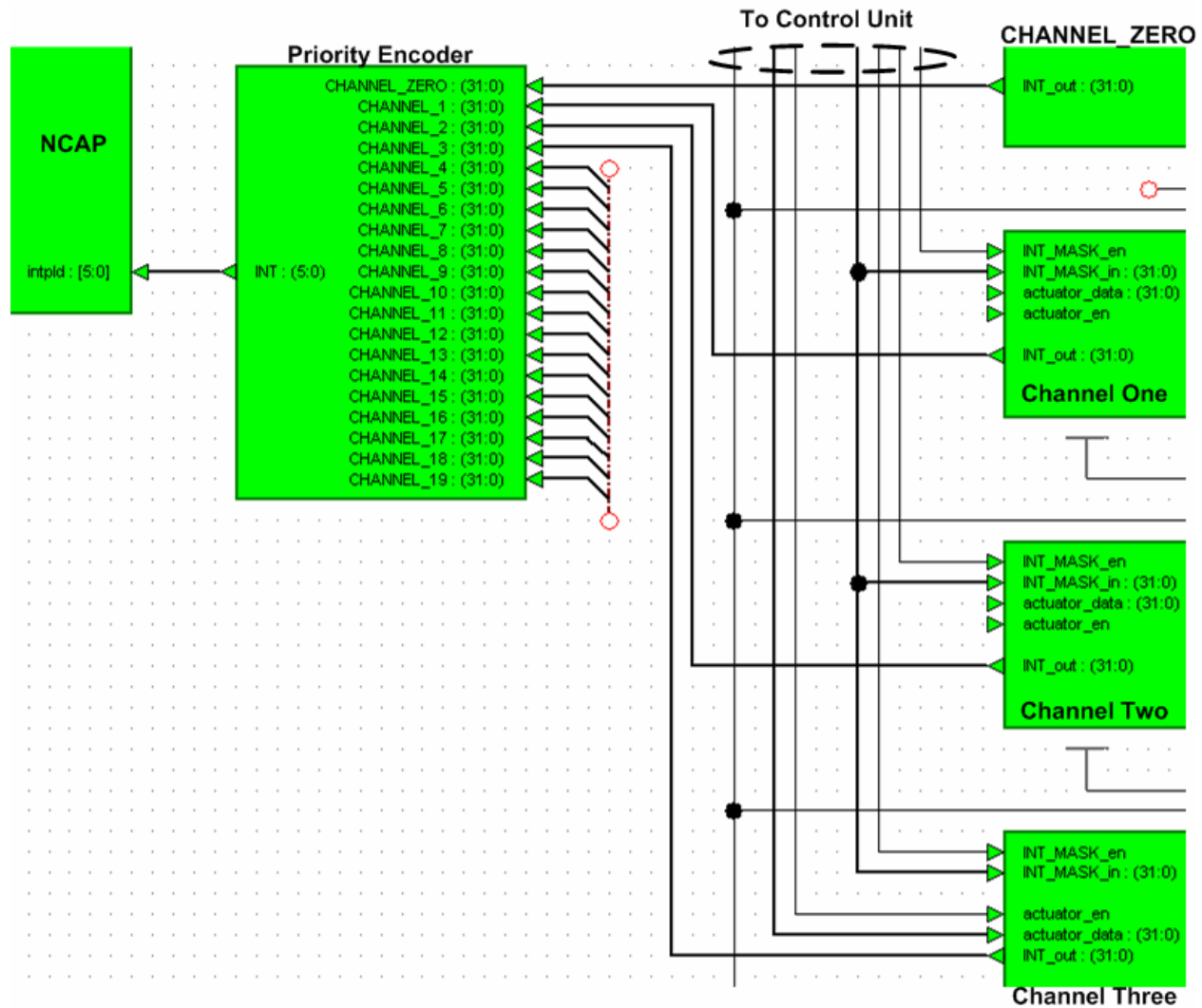


Figure 6-12 Priority encoder connections with Transducer channels, part (b) of Figure 6-10

For this interface, we simply needed to connect the generated interrupt from the transducer channels (AND operation between the status and interrupt mask registers) with the priority encoder. Then, the priority encoder encoded these generated interrupts onto six bits so that it could be interpreted by the embedded stripe's interrupt controller. Also, channels four through 19 were grounded so that they do not affect the behavior of the priority encoder that was previously designed in Section 5.2.4. Next we show the interface between the TIM control unit and the transducer channels in Figure 6-13.

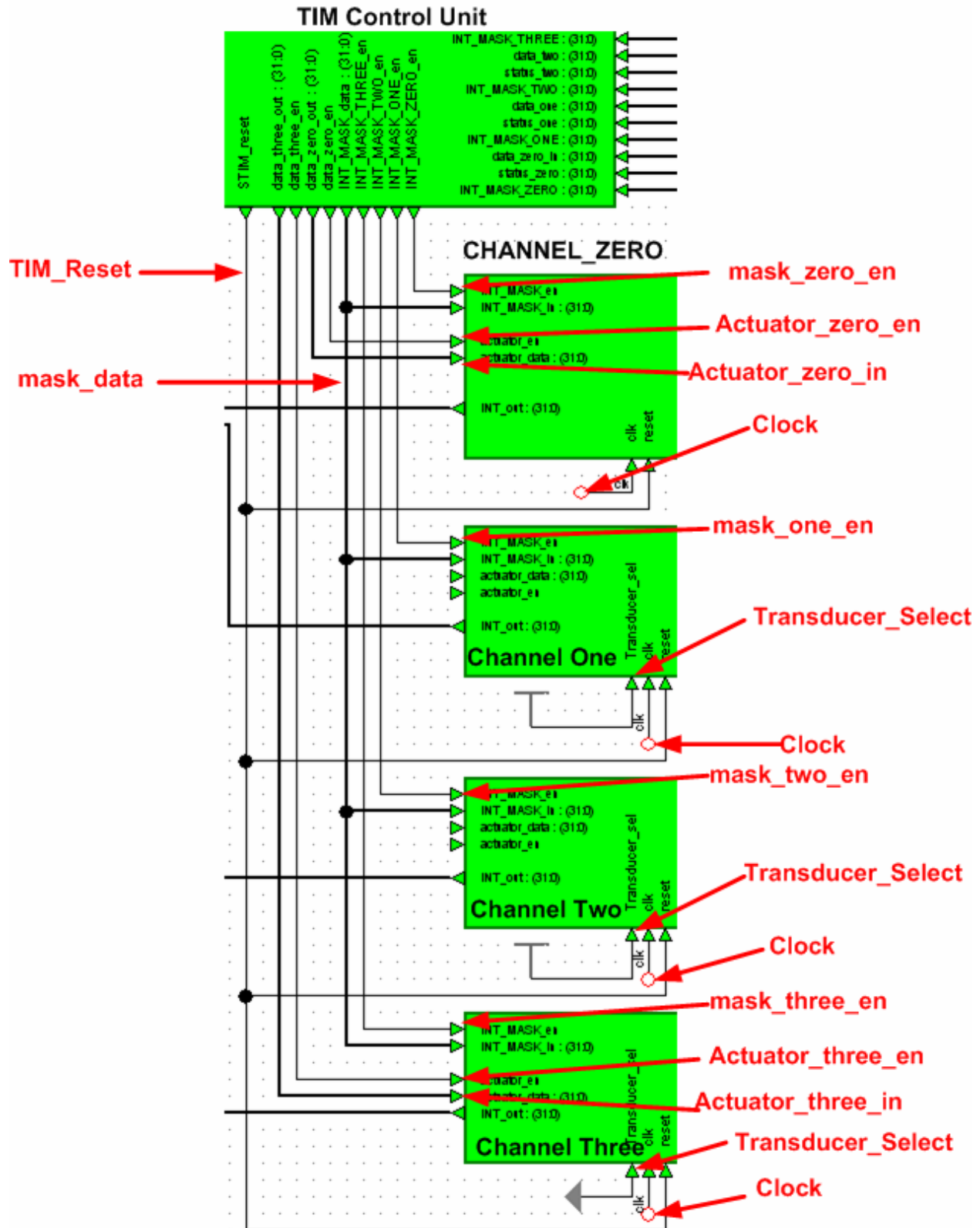


Figure 6-13 Transducer Channel Connections with Control Logic, part(c) of Figure 6-10

In this figure we show the connections for the inputs of the transducer channels. Following the example that was shown in the Design Chapter, we used the TIM reset signal (generated by the control logic) as the reset signal for the transducer channel blocks. Also, since the control unit generates the control/data signals of the interrupt mask and actuator data register, we interfaced these signals (generated by the control unit) with the transducer channels. Note that only CHANNEL_ZERO and Channel three had actuator data, since the LED actuators were implemented in Channel three, and CHANNEL_ZERO needs to represent every implemented transducer. The *Transducer_Sel* signal of the individual transducer channel blocks is used to configure the individual channel for a sensor or actuator configuration (Figure 5-51). Therefore, for channels one and two we grounded this signal (because those channels represented the sensors), and we connected the select signal to power for channel three (actuator channel).

Next, we show the interface between the transducer channels and their respective “glue” logic in Figure 6-14. The connections between the channels and the “glue” logic were done using the reference example design that was shown in Section 5.2.3. Therefore, the “glue” logic generated the status register’s control/data signals of every transducer channel. Then, for the actuator channels, it used the output of the actuator channel’s data register (CHANNEL_ZERO and Channel three) and used this value to set the actuator upon reception of a trigger command. It is important to note that for global triggers the actuator is set with the data set that is stored on CHANNEL_ZERO’s data register. Therefore, we used a MUX to select between the data from CHANNEL_ZERO or the data from Channel three. Then, we used the global trigger signal (*trigger_zero* signal generated by the control unit) as the select line for this MUX.

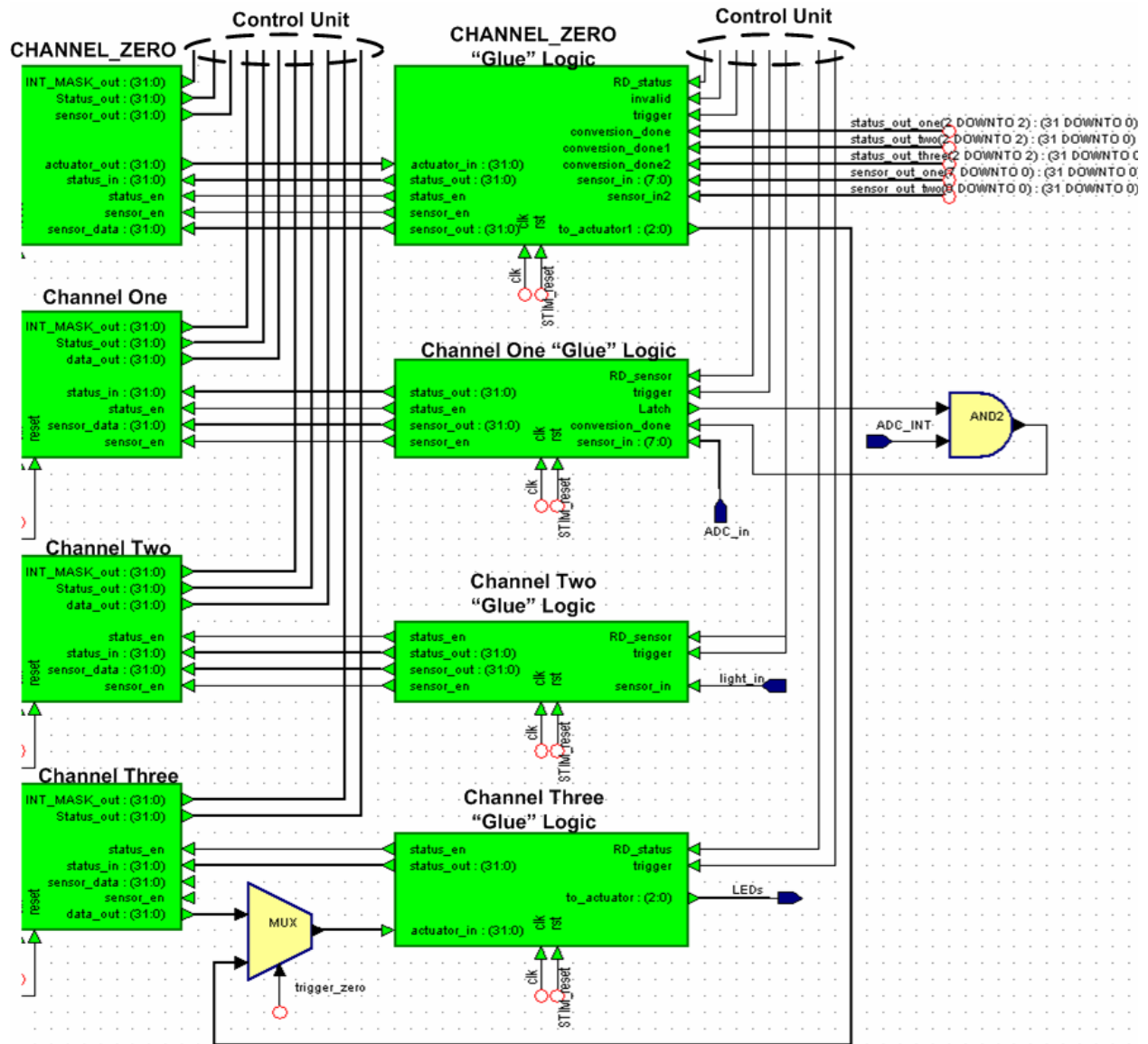


Figure 6-14 Transducer channels and glue logic, part (d) of Figure 6-10

Figure 6-15 shows the interface between the control unit, transducer channels, and “glue” logic. For this interface, we sent the output of the transducer channel registers (data, status and interrupt mask registers) to the control unit so that they could be read by the NCAP. Then, the control unit sent the trigger and status interface signals to the “glue” logic so that the transducer could be triggered, and the proper status control/data bits of the status registers could be generated.

TIM Control Unit

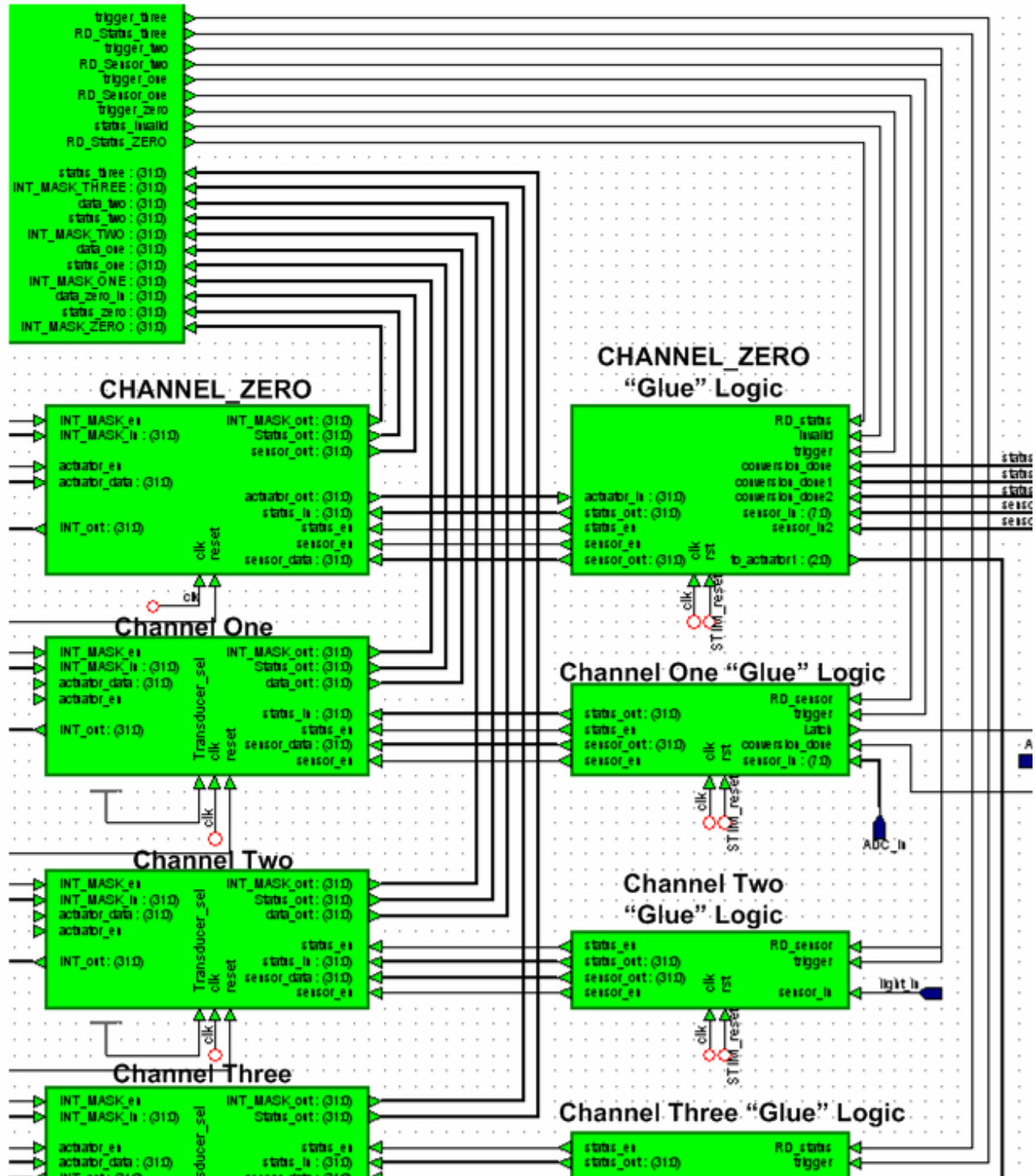


Figure 6-15 Interface between Control Unit, Transducer Channels and “glue” logic, part (e) of Figure 6-10

Next, we discuss the software interrupt handlers for the TIM interrupts. In order to do this, we first discuss the possible generated interrupts by the transducer channels. The Trigger

ACK interrupt is generated when the channel has been completely sampled/set and it is cleared when the status or data register is read. Note that for sensors this bit is cleared when the data register is read, and for actuators this is cleared when the status register is read.

The Invalid Command interrupt is generated by CHANNEL_ZERO when the NCAP tries to access a command that is not implemented or invalid and this bit is cleared when the status register is read.

The TIM/Channel Operational interrupt occurs upon power-up, since as has been mentioned before that the TIM and its channels is operational upon power-up. Therefore, the interrupt handler simply needs to clear the interrupt mask of the channel that generated the interrupt.

The Corrections enabled/disabled interrupt never occurs, since the TIM does not have capabilities to apply corrections to the transducer data. However, we set up the routine to clear the mask, read the status register and then set the mask again.

The different interrupt handler routines that have been described for this implementation are summarized in Table 6-2.

In this section we have shown the complete implementation for a proof of concept NCAP/TIM combination, in which the TIM was interfaced to three transducers and an Ethernet network was configured to interface with the NCAP. Next, we will show the tests and results of our implementation.

Table 6-2 Interrupt Service Routines summary

Interrupt	Handler Routine
Trigger ACK	CHANNEL_ZERO: 1. Clear the interrupt mask 2. Read the transducer channel data 3. Set the interrupt mask. Channel 1: 1. Clear the interrupt mask 2. Read the transducer channel data 3. Set the interrupt mask Channel 2: 1. Clear the interrupt mask 2. Read the transducer channel data 3. Set the interrupt mask Channel 3: 1. Clear the interrupt mask 2. Read Status Register and signal that the channel completely acquired the data 3. Set the interrupt mask
Invalid Command	1. Clear the interrupt mask 2. Read CHANNEL_ZERO status register 3. Set the interrupt mask
TIM/Channel Operational	Clear the interrupt mask
Corrections enabled/disabled	Will never occur but just in case clear the interrupt mask, read the status register and set the interrupt mask.

6.3 TEST RESULTS

The test results for the system implementation can be divided in two categories, hardware and software. For the hardware results we can take a look at the FPGA synthesis results which were efficient for the implementation, as it performed well (speed) and it did not consume too many resources. Table 6-3 summarizes the synthesis results.

Table 6-3 Synthesis Results

Area Usage (LEs)	FPGA IO Usage	Clock Rate
415/4160 (9%)	14 out of 40 (35%)	78.01 MHz

Note that the area usage was rather small which has to do with the fact that the hardware consisted of registers, a control unit (state machine) and combinational logic. The IO usage consisted of 10 pins used for the ADC (eight for output and two for control), one pin was used for the light sensor, and the LEDs used three I/O bits.

The other test results include the software simulation, which tested the complete command set of the TIM as well as the top level functionality of the NCAP. The tests mentioned in the Specifications Chapter were applied and checkpoints were set up in software to test the functionality. The debugging was done by using printf statements through hyperterminal. Each of the three major steps in the system's tests was executed yielding positive results.

These steps consisted of testing system initialization (NCAP objects are instantiated and initialized, and TEDS is read transducer-specific objects are instantiated and initialized). Then, we applied an individual trigger to every implemented TIM channel and sent/received a dummy packet over the network using the client/server model. The last test consisted of applying a global trigger command to the TIM and sending/receiving a dummy packet over the network using the publish/subscribe communication model.

Next, we will show the hyperterminal output along with an explanation about each of the three tests described above. Figure 6-16 shows the output that was generated when the NCAP/TIM combination went through the initialization process. The five steps that initialize the system and generate the output in the Figure are as follows:

1. Reset the TIM in order to test that TIM command. This is done by an application-specific call to the *Reset* TIM command that was previously shown in Table 5-3.
2. Read Meta-TEDS information by calling the *GetNumberOfTransducerChannels* and *GetMinimumSamplingPeriod* respectively. This information is stored and the

individual channels are instantiated. Then each individual transducer channel calls the *GetMetadata* operation to read each Channel-TEDS so that the application knows what type of transducer it is.

3. The different NCAP objects are initialized by means of the *Initialize* and *GoActive* operations of the Block Class.
4. The application calls the *irq_init()* function which is application-specific and initializes the interrupt controller by setting the interrupt masks as well as setting the operation mode to 6-bit priority.
5. The Ethernet MAC/PHY chip is initialized by calling the *smc_init()* function that is provided by the network library.

```
board - HyperTerminal
File Edit View Call Transfer Help

*****
**                                     **
**               IEEE 1451 Demonstration               **
**                                     **
**                                     **
**                                     **
**                                     **
*****

Reset the TIM ← 1
Reading TEDS... ← 2
System has 3 transducers according to the TEDS
TEDS has been completely read and the necessary objects have been instantiated

Initializing Object Classes ← 3
NCAP is operational!!!
Sending Interrupt mask register information to the TIM ← 4
Initializing SMSC Ethernet MAC controller chip ← 5
Complete System is operational!!

_

Connected 3:44:46  Auto detect  38400 8-N-1  SCROLL  CAPS  NUM  Capture  Print echo
```

Figure 6-16 Initialization Results

After verifying that all these operations occur correctly, the application system is said to be operational. The next test shows the individual triggering of every implemented TIM channel. The application-specific code that was used for this test is shown in Figure 6-17.

```
(1) check = UpdateAndRead(temperature, out);
if (check != 0)
    printf("Trigger did not execute correctly\r\n");
else
    printf("Channel 1 executed succesfully\r\n");

(2) check = UpdateAndRead(light, out_1);
if (check != 0)
    printf("Trigger did not execute correctly\r\n");
else
    printf("Channel 1 executed succesfully\r\n");

(3) check = WriteAndUpdate(LEDs, work);
if (check != 0)
    printf("Trigger did not execute correctly\r\n");
else
    printf("Channel 1 executed succesfully\r\n");

(4) server = Execute(client_port,
                    0,
                    0,
                    server_input,
                    server_output);
if (server != 0)
    printf("Error in communication\r\n");
else
    printf("Completed the client-side communication succesfully\r\n");
```

Figure 6-17 Application Code for Individual Triggers and Client-Server Operations

The code shown in Figure 6-17 can be explained as follows:

1. Channel 1 is triggered. When this is issued to the TIM, it responds by sending the ACK as is shown in the output. After this ACK is received the raw data of the sensor is read and within the same function a local call to the *ConvertTemperature* operation is executed to give the SI units of the sensor reading. The steps mentioned after the issue of the trigger command are transparent to the application.

2. The next step that is done is triggering channel two to see if there is light or not in the room. After receiving the ACK signal the sensor reading was read. Note that there is no conversion needed since this information was interpreted as a Boolean (TRUE = light, FALSE = dark).
3. The third step was to trigger the actuator with the information from the light and the temperature sensor. Since the temperature was operating in “normal” range LED 1 was set, and because there was no light there were no other LEDs that were set in the system. Note that this function also generated a Trigger ACK
4. The *Execute* function was called by the Client Port. Within this function, the dummy test packet was sent to the network infrastructure by means of the *MarshalAndTransmitReceive* function. This function then executes the loopback test that consists of sending and receiving the complete packet information in the Ethernet MAC/PHY chip. Note that the two sent and received packets matched as is shown in the Figure.

The result of this test is shown in Figure 6-18.

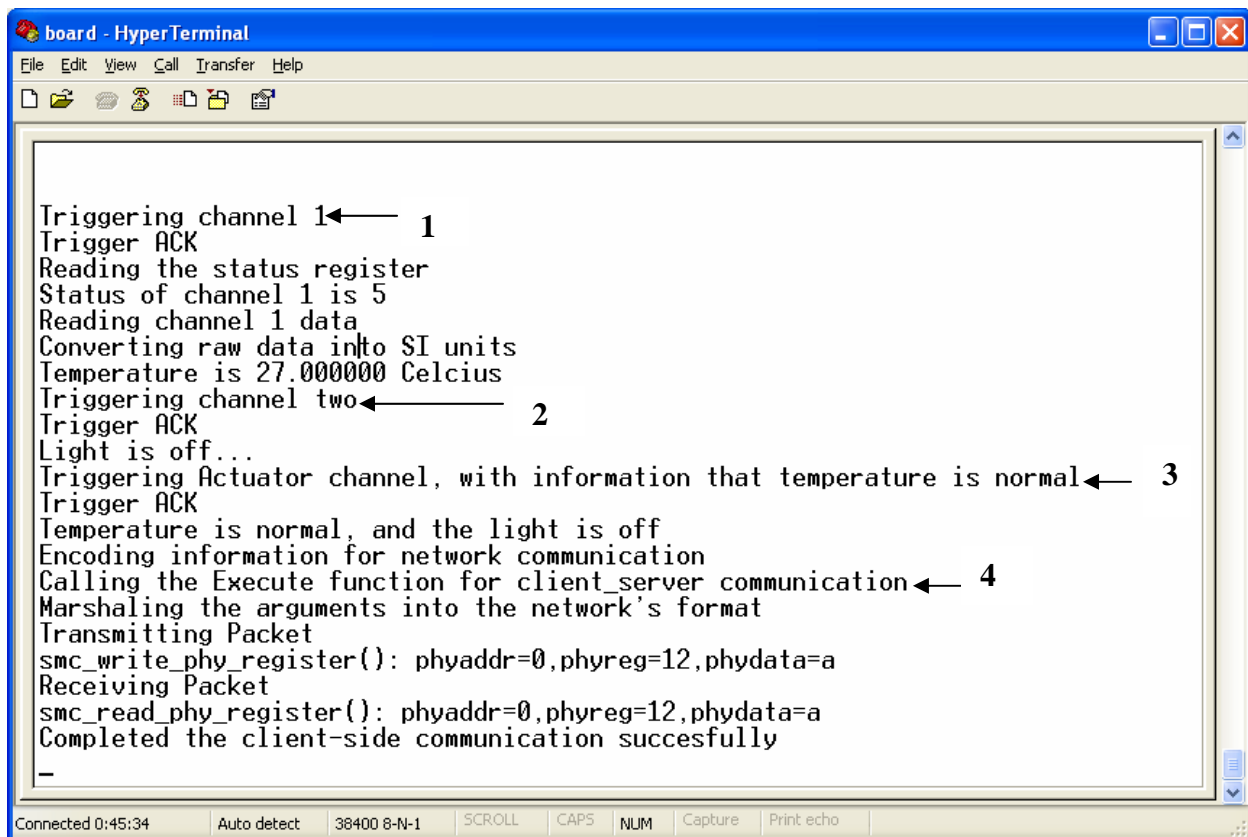
The next test consisted of the last iteration through the program. This iteration tests global triggering and publishes the information over the network. The code that was used for this test is shown in Figure 6-19.

The code (Figure 6-19) shows how the *UpdateAll* function is called to apply a global trigger command to the TIM. The interaction shown in the figure can be described as follows:

1. Every implemented channel is triggered. Note that each individual channel generates its own trigger ACK and the NCAP then reads the individual channels first and outputs the information, and at the end it reads CHANNEL_ZERO,

which contains all the information about all the transducer channels. Note that the output from the individual channels and the CHANNEL_ZERO match.

2. The *Publish* operation was called by the Publisher Port. Within this function the *MarshalAndTransmitReceive* operation was called to send the test packet information over the network. Note that this time the register that was used was 0xA, and the packet that was written was 0x1F90 which again matched the sent and received packages proving that the operation completed successfully.



```
board - HyperTerminal
File Edit View Call Transfer Help
Triggering channel 1 ← 1
Trigger ACK
Reading the status register
Status of channel 1 is 5
Reading channel 1 data
Converting raw data into SI units
Temperature is 27.000000 Celcius
Triggering channel two ← 2
Trigger ACK
Light is off...
Triggering Actuator channel, with information that temperature is normal ← 3
Trigger ACK
Temperature is normal, and the light is off
Encoding information for network communication
Calling the Execute function for client_server communication ← 4
Marshaling the arguments into the network's format
Transmitting Packet
smc_write_phy_register(): phyaddr=0,phyreg=12,phydata=a
Receiving Packet
smc_read_phy_register(): phyaddr=0,phyreg=12,phydata=a
Completed the client-side communication succesfully
_
Connected 0:45:34 Auto detect 38400 8-N-1 SCROLL CAPS NUM Capture Print echo
```

Figure 6-18 Main Loop first iteration

```

(1) check = UpdateAll(transducer_block);
if (check != 0)
    printf("Error in Global Trigger\r\n");
(2) check = Publish(publisher_port, data);
if (check != 0)
    printf("Error in communication\r\n");
else
    printf("Information Published\r\n");

```

Figure 6-19 Application Code for Global Trigger and Publish-Subscribe Operations

The result of this test is shown in Figure 6-20.

```

board - HyperTerminal
File Edit View Call Transfer Help
Applying a global trigger ← 1
Channel 2 Trigger ACK |
Light is on...
Channel 3 Trigger ACK
Channel 1 Trigger ACK
Temperature is 26.000000 Celcius
CHANNEL_ZERO Trigger ACK
Light is on...
Temperature is normal, and light is on
Temperature is 26.000000 Celcius
Encoding information for network communication
Publishing the information in the network ← 2
Transmitting Packet
smc_write_phy_register(): phyaddr=0,phyreg=a,phydata=1f90
Receiving Packet
smc_read_phy_register(): phyaddr=0,phyreg=a,phydata=1f90
Information published
Connected 0:46:24 Auto detect 38400 8-N-1 SCROLL CAPS NUM Capture Print echo

```

Figure 6-20 Main Loop Second Iteration

In this chapter we have shown the proof of concept application that was implemented in order to test the design of our NCAP/TIM architecture. In the next chapter, we will summarize

the efforts of this thesis, discuss the conclusions, and give the future directions that this project could undertake.

7.0 CONCLUSIONS AND FUTURE WORK

In this thesis we have presented a single chip solution for an IEEE 1451 compliant system using the following structure. First, we introduced the standard, and gave background information about its structure and functionality in Chapter 1.0. Then, in Chapter 2.0, we showed a comprehensive look at the problem where previous solutions were discussed and our solution was proposed. In Chapter 3.0, we derived a set of requirements for the system in order to remain compliant with the standard. From these requirements, we derived a set of specifications and tests as was depicted by Chapter 4.0. Next, in Chapter 5.0, the different objects in the system were designed using the specifications previously defined. In Chapter 6.0, we showed a proof of concept implementation along with the test results that showed the system's performance and compliance with the standard.

7.1 CONCLUSIONS

In this thesis, we have presented the design of an NCAP/TIM combination that will aid in the development and understanding of the IEEE 1451 family of standards. The following discussion points out the key aspects of the design that was presented in this thesis:

- By meeting the high-level objectives (high-level of configurability, better performance, etc.) of this design, we have created a flexible and high-performance architecture that can be reused for control networks.
- Eliminating the TII in favor of a parallel high performance bus connection improves the speed performance of the NCAP/TIM communication. This speed improvement is significant since the sampling rate for a system with 10 transducers (5 sensors and 5 actuators) would be 374 KHz (assuming an ADC conversion rate of 2.5 μ Sec), as opposed to the 51 Hz that the 1451.2 connection yields under the same configuration.
- The low power capabilities of the ARM processor and the high-speed connection of the NCAP and TIM make this solution helpful for application systems that contain a large number of transducers that need to be sampled periodically in a small amount of time.
- This solution will provide an efficient alternative to the widely criticized TII communication protocol of the IEEE 1451.2 standard.

7.2 FUTURE WORK

Future work for the presented solution consists of the following:

- Complete an Application Specific Integrated Circuit (ASIC) for this IEEE 1451 single chip solution.

- Design a completely re-configurable TIM block. This proposed block would have a database of TEDS associated with it that contains information about a variety of transducers. Then, using this database we can select the transducer(s) that we would like to implement, and the complete transducer channel TIM block would be instantiated. In order to accomplish this, we would need to design a Graphical User Interface (GUI) that has the capability of dynamically setting the I/O ports of the control unit as well as the individual transducer channels. Note that designers would still have to build the “glue” logic with the physical transducer.

APPENDIX A. NCAP OBJECT MODEL

This appendix discusses the top-level structures (using UML class diagrams), and a brief explanation for the standard-defined object classes that were not described in the specifications chapter. First, we show the Root class which is the root for the class hierarchy of all objects defined in the standard. Its structure is shown below.

IEEE1451_Root
+_class_ID : ClassID +_description : String +_class_name : String +GetClassName(out class_name : String) : OpReturnCode +GetClassID(out class_id : String) : OpReturnCode

Next, we describe the Base Transducer Block class. This class is the root for the class hierarchy of all Transducer Block Objects. The structure for this class is shown below.

IEEE1451_BaseTransducerBlock
+_class_ID : ClassID +_description : String +_class_name : String +IORead(in io_input_arguments : ArgumentArray, out io_output_arguments : ArgumentArray) : OpReturnCode +IOWrite(in io_input_arguments : ArgumentArray, out io_output_arguments : ArgumentArray) : OpReturnCode +SetIOControl(in control_arguments : ArgumentArray) : OpReturnCode +GetIOStatus(in io_input_arguments : ArgumentArray, out status : ArgumentArray) : OpReturnCode

The next class is the Component class (shown below), which is the root for the class hierarchy of all Component Objects.

IEEE1451_Component
+_class_ID : ClassID +_description : String +_class_name : String +SpecifyRuleBasis(in rule_basis : ushort) : OpReturnCode

Now, we define the Parameter class which is used to model network visible variables and to provide a means for accessing them. This class' structure is shown in the next figure.

IEEE1451_Parameter
+_class_ID : ClassID +_description : String +_class_name : String
+Read(out data : ArgumentArray) : OpReturnCode +Write(in data : ArgumentArray) : OpReturnCode

The next class is the Scalar Parameter class. This class is used to model physical world quantities that do not have dimensions or orientation with them, and are appropriately represented as mathematical scalars. The structure for this class is shown below.

IEEE1451_ScalarParameter
+_class_ID : ClassID +_description : String +_class_name : String
+GetDataType(out value_datatype : ushort) : OpReturnCode +GetUnits(out value_units : Units) : OpReturnCode +SetDataType(out value_datatype : ushort) : OpReturnCode +SetUnits(in value_units : Units) : OpReturnCode

Next, we define the Scalar Series Parameter class which is used to model physical world quantities, best modeled as a succession of scalars evenly distributed along some dimension. Examples of such quantities are time series, fourier transforms, and mass spectra. This class' structure is shown below.

IEEE1451_ScalarSeriesParameter
+_class_ID : ClassID +_description : String +_class_name : String
+GetAbscissaUnits(out abscissa_units : Units) : OpReturnCode +GetAbscissaIncrement(out abscissa_increment : Argument) : OpReturnCode +GetAbscissaOrigin(out abscissa_origin : Argument) : OpReturnCode +SetAbscissaUnits(in abscissa_units : Units) : OpReturnCode +SetAbscissaIncrement(in abscissa_increment : Argument) : OpReturnCode +SetAbscissaOrigin(in abscissa_origin : Argument) : OpReturnCode

We now define the Vector Parameter class which is used to model physical world quantities that have multiple dimensions and perhaps orientations associated with them, and are appropriately represented as mathematical vectors. The structure for this class is shown in the following figure.

IEEE1451_VectorParameter
+_class_ID : ClassID +_description : String +_class_name : String
+GetDimension(out dimension : ushort) : OpReturnCode +GetDatatype(out value_datatype : ushort) : OpReturnCode +GetUnits(out value_units : UnitsArray) : OpReturnCode +SetDimension(in dimension : ushort) : OpReturnCode +SetDatatype(in value_datatype : ushort) : OpReturnCode +SetUnits(in value_units : UnitsArray) : OpReturnCode

Next, we show the Vector Series Parameter which is used to model a uniform series of physical world quantities that have dimensions and orientation associated with them and are appropriately represented as mathematical vectors. This class' structure is shown below.

IEEE1451_VectorSeriesParameter
+_class_ID : ClassID +_description : String +_class_name : String
+GetAbscissaUnits(out abscissa_units : Units) : OpReturnCode +GetAbscissaIncrement(out abscissa_increment : Argument) : OpReturnCode +GetAbscissaOrigin(out abscissa_increment : Argument) : OpReturnCode +SetAbscissaUnits(in abscissa_units : Units) : OpReturnCode +SetAbscissaIncrement(in abscissa_origin : Argument) : OpReturnCode +SetAbscissaOrigin(in abscissa_increment : Argument) : OpReturnCode

We now define the Time Parameter class which is used to represent time parametric values. This class' purpose is to model network visible variables that directly or indirectly represent the time of some event, or the duration between two events, where the significant characteristic of the event is the time rather than some other value. The structure for this class is shown below.

IEEE1451_TimeParameter
+_class_ID : ClassID +_description : String +_class_name : String
+GetUncertainty(out uncertainty : Uncertainty) : OpReturnCode +GetTimeType(out time_type : ushort) : OpReturnCode +GetEpochRepresentation(out epoch_representation : ushort) : OpReturnCode +GetEpoch(out epoch : TimeRepresentation) : OpReturnCode #RegisterNotifyOnUpdate(in notification_operation, in registration_id : ushort) : OpReturnCode #DeregisterNotifyOnUpdate(in notification_operation, in registration_id : ushort) : OpReturnCode

The next class is the Action class. This class is used to represent activities that alter system state and that require significant time to execute compared to other activities in the system. The structure for this class is shown below.

IEEE1451_Action
+_class_ID : ClassID +_description : String +_class_name : String
+GetActionFailedTimeoutDuration(out action_failed_timeout_duration : TimeRepresentation) : OpReturnCode +InvokeTransaction(in input_arguments : ArgumentArray, out transaction_id : ushort) : OpReturnCode +GetActionState(out action_state : ushort) : OpReturnCode +AbortTransaction(in transaction_id : ushort) : OpReturnCode +ReleaseTransaction(in transaction_id : ushort) : OpReturnCode +ForceAcquireTransaction(out new_transaction_id : ushort) : OpReturnCode

Next is the File class which is an abstraction of a data resource. This class represents a block of memory, which may be opened, closed, read from, and written to. The structure for this class is shown below.

IEEE1451_File
+_class_ID : ClassID +_description : String +_class_name : String
+OpenForRead(out transaction_id : ushort, out actual_image_size : uint) : OpReturnCode +OpenForWrite(out transaction_id : ushort, out max_file_size : uint) : OpReturnCode +Read(in transaction_id : ushort, in requested_number_of_octets : uint, out actual_number_of_octets_read : uint, out data : OctetArray) : OpReturnCode +Write(in transaction_id : ushort, in requested_number_of_octets : uint, in data : OctetArray, out actual_number_of_octets_written : uint) : OpReturnCode +Close(in transaction_id : ushort) : OpReturnCode +ForceClose() : OpReturnCode +GetFileState(out file_state : ushort, out actual_image_size : uint, out max_file_size : uint) : OpReturnCode

We now show the Partitioned File class. This class is used for Files that are subdivided into a number of partitions. Its structure is shown below.

IEEE1451_PartitionedFile
+_class_ID : ClassID +_description : String +_class_name : String
+SeekPartition(in transaction_id : ushort, in partition_id : uint, out actual_partition_image_size : uint, out max_partition_size : uint) : OpReturnCode +GetCurrentPartition(in transaction_id : ushort, out partition_id : uint, out actual_partition_image_size : uint, out max_partition_size : uint) : OpReturnCode +GetNumberOfPartitions(in transaction_id : ushort, out number_of_partitions : ushort) : OpReturnCode +GetPartitionedFileSubstate(in transaction_id : ushort, out partition_state : ushort) : OpReturnCode

Up next, we have the Component Group class which provides a way to specify set membership relations between objects in a system. This class' structure is depicted by the figure shown below.

IEEE1451_ComponentGroup
+_class_ID : ClassID +_description : String +_class_name : String
+AddMember(in member_properties : ObjectProperties) : OpReturnCode +DeleteMember(in member_properties : ObjectProperties) : OpReturnCode +GetMembers(out members_properties : ObjectPropertiesArray) : OpReturnCode +SetMembers(in members_properties : ObjectPropertiesArray) : OpReturnCode +LookupMembersByName(in query_object_name : String, out member_properties : ObjectPropertiesArray) : OpReturnCode +LookupMemberByDispatchAddress(in query_dispatch_address : ObjectDispatchAddress, out member_properties : ObjectPropertiesArray) : OpReturnCode +GetNumberOfMembers(out number_of_members : ushort) : OpReturnCode +GetNextMember(in first_flag : bool, out member_properties : ObjectProperties) : OpReturnCode +CancelIteration() : OpReturnCode +LookUpMemberByObjectTag(in query_object_tag : ObjectTag, out member_properties : ObjectPropertiesArray) : OpReturnCode

The next class is the Service class. This class is the root for the class hierarchy of all service objects. The service classes represent object types used to support communication and other aspects of block functionality. The structure for this class is shown below.

IEEE1451_Service
+_class_ID : ClassID +_description : String +_class_name : String
+SpecifyRuleBasis(in rule_basis : ushort) : OpReturnCode

Next, we show the Base Port class which is the root for the class hierarchy of all communications port objects used to send communications via the underlying network. Its structure is shown below.

IEEE1451_BasePort
+_class_ID : ClassID +_description : String +_class_name : String
+SetTimeout(in client_timeout : TimeRepresentation) : OpReturnCode +GetTimeout(out client_timeout : TimeRepresentation) : OpReturnCode +SetMessagePriority(in message_priority : ushort) : OpReturnCode +GetMessagePriority(in message_priority : ushort) : OpReturnCode

We now show the Base Client Port class. This class is the root for the class hierarchy of all client-server communication client-side port objects. The structure for this class is shown below.

IEEE1451_BaseClientPort
+_class_ID : ClassID +_description : String +_class_name : String
+SetServerObjectTag(in server_object_tag : ObjectTag) : OpReturnCode +GetServerObjectTag(out server_object_tag : ObjectTag) : OpReturnCode

Up next we have the Asynchronous Client Port class. This class provides client-side functionality for an asynchronous, non-blocking, client-server communication model. The structure for this class is shown below.

IEEE1451_AynchronousClientPort
+_class_ID : ClassID +_description : String +_class_name : String
#ExecuteAsynchronous(in server_operation_id : ushort, in server_input_arguments : ArgumentArray, out transaction_id : ushort) : OpReturnCode #GetAsynchronous(in transaction_id : ushort, out result_status : ushort) : OpReturnCode #GetResult(in transaction_id : ushort, out server_output_arguments : ArgumentArray) : OpReturnCode #AbortTransaction(in transaction_id : uint) : OpReturnCode #ForceAcquireTransaction(out new_transaction_id : ushort, out current_server_operation_id : ushort) : OpReturnCode

The next class is the Base Publisher Port class which provides basic publisher-side for its subclasses. This class' structure is shown below.

IEEE1451_BasePublisherPort
+_class_ID : ClassID +_description : String +_class_name : String
+SetPublicationTopic(in publication_topic : PublicationTopic) : OpReturnCode +GetPublicationTopic(out publication_topic : PublicationTopic) : OpReturnCode +GetPublicationKey(out publication_key : ushort) : OpReturnCode +SetPublicationDomain(in publication_domain : PubSubDomain) : OpReturnCode +GetPublicationDomain(out publication_domain : PubSubDomain) : OpReturnCode

Next, we describe the Self Identifying Publisher Port class. This class provides publisher-subscribe communication model with operations to allow the subscriber to establish communication with the publisher, and the publisher to notify subscribers of changes in the publication policy. The structure for this class is shown below.

IEEE1451_SelfIdentifyingPublisherPort
+_class_ID : ClassID +_description : String +_class_name : String
+GetPublisherMetadata(in publication_id : ushort, in cached_publication_change_id : ushort, out publisher_metadata : ArgumentArray) : OpReturnCode +PublishPublisherMetadata(in publication_id : ushort, in cached_publication_change_id : ushort) : OpReturnCode #PublishWithIdentification(in publication_contents : ArgumentArray) : OpReturnCode #RegisterPublisher(in callback_operation_reference, out publication_id : ushort) : OpReturnCode #DeregisterPublisher(in callback_operation_reference, in publication_id : ushort) : OpReturnCode

Now we describe the Event Generator Publisher Port class which is used to allow events internal to the operation of a block to result in the publication of an event record. This class' structure is shown in the following figure.

IEEE1451_EventGeneratorPublisherPort
+_class_ID : ClassID +_description : String +_class_name : String
+GetEventGeneratorState(out event_generator_state : ushort) : OpReturnCode +SetEventSequenceNumber(in event_sequence_number : uint) : OpReturnCode +GetEventSequenceNumber(out event_sequence_number : uint) : OpReturnCode +SetResponseTag(in response_object_tag : ObjectTag) : OpReturnCode +GetLastTimestamp(out last_event_timestamp : TimeRepresentation) : OpReturnCode +EnableEventPublication() : OpReturnCode +DisableEventPublication() : OpReturnCode +GetResponseTag(out response_object_tag : ObjectTag) : OpReturnCode

The next class is the Mutex Service class. This class (shown below) provides mutual exclusion capability.

IEEE1451_MutexService
+_class_ID : ClassID +_description : String +_class_name : String
+Lock(in timeout : TimeRepresentation, out transaction_id : ushort) : OpReturnCode +Unlock(in transaction_id : ushort) : OpReturnCode +TryLock(out transaction_id : ushort) : OpReturnCode +IsLocked(out mutex_status : bool) : OpReturnCode +ForceAcquireLock(out transaction_id : ushort) : OpReturnCode

Lastly, we show the Condition Variable class. This class provides the capability for ordering concurrent activities. The structure for this class is shown in the following figure.

IEEE1451_ConditionVariableService
+_class_ID : ClassID +_description : String +_class_name : String
+Lock(in lock_timeout : TimeRepresentation, out transaction_id : ushort) : OpReturnCode +Unlock(in transaction_id : ushort) : OpReturnCode +TryLock(out transaction_id : ushort) : OpReturnCode +SetPredicateState(in predicate_state : bool, in transaction_id : ushort) : OpReturnCode +Wait(in wait_timeout : TimeRepresentation, out transaction_id : ushort) : OpReturnCode +SignalOne(in transaction_id : ushort) : OpReturnCode +SignalAll(in transaction_id : ushort) : OpReturnCode +IsLocked(out mutex_status : bool) : OpReturnCode +GetPredicateState(out predicate_state : bool) : OpReturnCode +ForceAcquireLock(out transaction_id : ushort) : OpReturnCode +ClearAllPendingWaits(in transaction_id : ushort) : OpReturnCode

APPENDIX B. NCAP SOFTWARE CODE

In this appendix, we show the software code that was written for the network and transducer access operations. The code is commented and shows what a user needs to modify for his/her particular application.

The code for the Execute operation is shown below.

```
// Execute function for the client side functionality of the client/server communication model
ClientServerReturnCode Execute(IEEE1451_ClientPort *a,
    /* in */ unsigned short execute_mode,
    /* in */ unsigned short server_operation_id,
    /* in */ ArgumentArray server_input_arguments,
    /* out */ ArgumentArray server_output_arguments
)
{
    if (execute_mode == 0){
        // Next line changes depending on the network infrastructure
        MarshalAndTransmitReceive(server_input_arguments, 0);
        // Timeout routine
        While (interrupt == 0){
            If (jiffies > timeout)
                Return 14;
            Else
                Return 0;
        }
    }
    else{
        // Next line changes depending on the network infrastructure
        MarshalAndTransmitReceive(server_input_arguments, 0);
        Return 0;
    }
}
```

It is important to note that *jiffies* is a timer interrupt signal that is used to make sure that the operation does not timeout.

Next is the software code for the Publish operation.

```
OpReturnCode Publish(IEEE1451_PublisherPort *a, ArgumentArray publication_contents){
    // Next line changes depending on the network infrastructure
```

```

    MarshalAndTransmtReceive(publication_contents, 1);
    return 0;
}

```

Now, we move on to the transducer access. We begin with the UpdateAndRead operation (used for sensor trigger).

```

OpReturnCode UpdateAndRead(IEEE1451_ParameterWithUpdate *a,
    /* out */ ArgumentArray data
)
{
    ArgumentArray temp;
    float *res;
    int raw;
    Boolean light;
    // First check to see what kind of sensor we're dealing with...
    // Check the object tags
    if (a->object_tag == "0.10") {

        // Temperature sensor
        // Trigger the sensor then read...
        // Next Line Changes depending on the transducer channel
        *STIM_ONE_TRIGGER(EXC_PLD_BLOCK0_BASE) = 1;
        temp->typeCode = FLOAT32_TC;
        // read the sensor after it was triggered
        // Timeout routine
        while (trigger_ACK != 1) {
            if (jiffies > channel1.ChannelUpdateTime)
                return 14;
        }
        // Next Line Changes depending on the transducer channel
        raw = *STIM_ONE_DATA(EXC_PLD_BLOCK0_BASE);
        ConvertTemperature(raw, res);
        temp->Arg_Union.float32Val = *res;
        return 0;
    }
    else if (a->object_tag == "0.11") {
        // Light Sensor
        // Trigger the sensor
        // Next Line Changes depending on the transducer channel
        *STIM_TWO_TRIGGER(EXC_PLD_BLOCK0_BASE) = 1;
        temp->typeCode = BOOLEAN_TC;
        // Read the sensor...
        while (trigger_ACK != 1) {
            if (jiffies > channel2.ChannelUpdateTime)
                return 14;
        }
        // Next Line Changes depending on the transducer channel
        light = *STIM_TWO_DATA(EXC_PLD_BLOCK0_BASE);
        temp->Arg_Union.booleanVal = light;
        return 0;
    }
    else {

```

```

        // LED Actuator so nothing should be done here...
        printf("No effect because reading from an actuator...\r\n");
        return 0;
    }
}

```

Note that this operation is written for the 3-Channel TIM that was implemented as the proof of concept design. However, as is shown by the comments it is easy to add more transducers for another application.

Next, is the *WriteAndUpdate* operation.

```

OpReturnCode WriteAndUpdate(IEEE1451_ParameterWithUpdate *a,
    /* in */ ArgumentArray data
)
{
    if (a->object_tag == "0.12") {
        // LED Actuator so write decode the argument array data and set the actuator
        Next Line Changes depending on the transducer channel
        *STIM_THREE_TRIGGER(EXC_PLD_BLOCK0_BASE) = data->Arg_Union.integer32Val;
        while (trigger_ACK != 1) {
            if (jiffies > channel2.ChannelUpdateTime)
                return 14;
        }
        return 0;
    }
    else { // Sensors so nothing should be done here...
        printf("No effect because writing to a sensor...\r\n");
        return 0;
    }
}

```

Next, we show the *UpdateAll* operation. It is important to note that this operation times out if the trigger ACK is not received within the *Worse Channel Update Time* that is found on the TIM's Meta-TEDS.

```

OpReturnCode UpdateAll(IEEE1451_TransducerBlock *a){
    // Set the global trigger function...
    printf("Applying a global trigger\r\n");
    *STIM_ZERO_TRIGGER(EXC_PLD_BLOCK0_BASE) = 1;

    while (trigger_ACK != 1) {
        if (jiffies >= CHANNEL_ZERO.worseChannelUpdateTime){
            printf("Error");
            return 14;
        }
    }
}

```

```
    }  
    return 0;  
}
```

APPENDIX C. TIM VHDL CODE

This appendix is dedicated for the VHDL code of the control unit (for the 3-Channel TIM), the Transducer Block and the Priority Encoder. First, we show the VHDL code of the control unit. Note that in the VHDL code we show with comments where the additions must be made to expand this control unit for an 8-Channel TIM implementation.

```
-- Created by Gustavo E. Lopez
-- Basic slave implementation on the AMBA bus that decodes the instruction
-- for the TIM implementation

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY slave_interface IS

    PORT (
        -- AHB interface
        hreset      : IN std_logic;
        hclock      : IN std_logic;
        hwrite      : IN std_logic;
        hsel        : IN std_logic;
        htrans      : IN std_logic_vector(1 downto 0);
        hsize       : IN std_logic_vector(1 downto 0);
        hburst      : IN std_logic_vector(2 downto 0);
        haddress    : IN std_logic_vector(31 downto 0);
        hwdata      : IN std_logic_vector(31 downto 0);
        hready      : OUT std_logic;
        hresp       : OUT std_logic_vector(1 downto 0);
        hrdata      : OUT std_logic_vector(31 downto 0);

        -- Inteface to the TIM
        -- Interrupt mask registers definition, along with their respective enable signals
        INT_MASK_ZERO      : IN std_logic_vector(31 downto 0);
        INT_MASK_ZERO_en   : OUT STD_LOGIC;
        INT_MASK_ONE       : IN std_logic_vector(31 downto 0);
        INT_MASK_ONE_en    : OUT STD_LOGIC;
        INT_MASK_TWO       : IN std_logic_vector(31 downto 0);
        INT_MASK_TWO_en    : OUT STD_LOGIC;
        INT_MASK_THREE     : IN std_logic_vector(31 downto 0);
        INT_MASK_THREE_en  : OUT STD_LOGIC;

        INT_MASK_data      : OUT std_logic_vector(31 downto 0);

        -- Status registers definition, along with its parcicular enable signals
```



```

status_zero          : IN std_logic_vector(31 downto 0);

status_one           : IN std_logic_vector(31 downto 0);
status_two           : IN std_logic_vector(31 downto 0);
status_three         : IN std_logic_vector(31 downto 0);

-- "Glue" Interface signals
RD_Status_ZERO : OUT std_logic;
RD_Sensor_one  : OUT std_logic;
RD_Sensor_two  : OUT std_logic;
RD_Status_three : OUT std_logic;
-- We would need to add the RD_Status/RD_Sensor signals for other transducers here

-- Define a signal that will signal the status register that an invalid command was sent from the NCAP
status_invalid      : OUT std_logic;

-- Channel Data definition, and control signals
-- Here is where we should define the control/data signals for the newer transducer data regs
data_zero_in        : IN std_logic_vector(31 downto 0);
data_zero_out        : OUT std_logic_vector(31 downto 0);
data_zero_en         : OUT std_logic;
data_one             : IN std_logic_vector(31 downto 0);
data_two             : IN std_logic_vector(31 downto 0);
data_three_en        : OUT std_logic;
data_three_out        : OUT std_logic_vector(31 downto 0);

-- Define signals that will be used to determine the trigger ACK
-- We would need to define five more trigger signals (for channels 4 through 8)
trigger_zero         : OUT std_logic;
trigger_one          : OUT std_logic;
trigger_two          : OUT std_logic;
trigger_three        : OUT std_logic;

-- Signal used to reset the entire TIM
STIM_reset           : OUT std_logic

);
END slave_interface;

```

ARCHITECTURE rtl OF slave_interface IS

```

SIGNAL haddress_reg      : std_logic_vector(31 downto 0);
SIGNAL hburst_reg        : std_logic_vector(2 downto 0);
SIGNAL htrans_reg        : std_logic_vector(1 downto 0);
SIGNAL hresp_reg         : std_logic_vector(1 downto 0);

-- Note that for "glue" logic/transducer channel interface output shown in the port declaration
-- We need to instantiate temporary signals right here...
SIGNAL mask_zero_en      : std_logic;
SIGNAL mask_one_en       : std_logic;
SIGNAL mask_two_en       : std_logic;
SIGNAL mask_three_en     : std_logic;
SIGNAL mask_data         : std_logic_vector(31 downto 0);

SIGNAL zero_out          : std_logic_vector(31 downto 0);

```

```

    SIGNAL zero_en          : std_logic;
    SIGNAL three_en         : std_logic;
    SIGNAL three_out        : std_logic_vector(31 downto 0);

    SIGNAL glue_zero        : std_logic;
    SIGNAL glue_one         : std_logic;
    SIGNAL glue_two         : std_logic;
    SIGNAL glue_three       : std_logic;

    SIGNAL invalid          : std_logic;

    SIGNAL zero_temp        : std_logic;
    SIGNAL one_temp         : std_logic;
    SIGNAL two_temp         : std_logic;
    SIGNAL three_temp       : std_logic;

    SIGNAL rst              : std_logic;

    SIGNAL internal_write    : std_logic;

    TYPE state_type IS (address,data);
    SIGNAL state : state_type;

BEGIN

-- We are always ready and we always respond with an OKAY responses to the initiating master.
hready <= '1';
hresp <= "00";

-- Create a FSM to control the internal read and write signals to the register bank.
PROCESS(hclock,hresetn)
BEGIN
    IF hresetn = '0' THEN
        internal_write <= '0';
        state <= address;
        invalid <= '0';
        zero_en <= '0';
        three_en <= '0';
        mask_zero_en <= '0';
        mask_one_en <= '0';
        mask_two_en <= '0';
        mask_three_en <= '0';
        glue_zero <= '0';
        glue_one <= '0';
        glue_two <= '0';
        glue_three <= '0';
        zero_temp <= '0';
        one_temp <= '0';
        two_temp <= '0';
        three_temp <= '0';

    ELSIF rising_edge(hclock) THEN
        CASE state IS
            WHEN address =>
                IF hsel = '1' AND htrans = "10" THEN
                    IF hwrite = '1' THEN

```

```

internal_write <= '1';
CASE haddress(15 downto 0) IS
  WHEN X"0000" =>
    mask_zero_en <= '1';
    mask_one_en  <= '0';
    mask_two_en  <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one  <= '0';
    glue_two  <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp  <= '0';
    two_temp  <= '0';
    three_temp <= '0';
    rst <= '1';
  WHEN X"0004" =>
    mask_zero_en <= '0';
    mask_one_en  <= '1';
    mask_two_en  <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one  <= '0';
    glue_two  <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp  <= '0';
    two_temp  <= '0';
    three_temp <= '0';
    rst <= '1';
  WHEN X"0008" =>
    mask_zero_en <= '0';
    mask_one_en  <= '0';
    mask_two_en  <= '1';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one  <= '0';
    glue_two  <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp  <= '0';
    two_temp  <= '0';
    three_temp <= '0';
    rst <= '1';
  WHEN X"000C" =>
    mask_zero_en <= '0';
    mask_one_en  <= '0';

```

```

mask_two_en  <= '0';
mask_three_en <= '1';
zero_en <= '0';
three_en <= '0';
invalid <= '0';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= '1';
-- We would add the Write INT_MASK commands for channels 4 through 8 here
WHEN X"2000" =>
    mask_zero_en <= '0';
    mask_one_en  <= '0';
    mask_two_en  <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '1';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';
WHEN X"2004" =>
    mask_zero_en <= '0';
    mask_one_en  <= '0';
    mask_two_en  <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '1';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';
WHEN X"2008" =>
    mask_zero_en <= '0';
    mask_one_en  <= '0';
    mask_two_en  <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';

```

```

invalid <= 'I';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= 'I';
WHEN X"200C" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= 'I';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= 'I';
-- Add the addresses for Channels 4 through 8 here, however this command will always
--return an invalid command since we cannot write to the Status register
WHEN X"4000" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= 'I';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= 'I';
    one_temp <= 'I';
    two_temp <= 'I';
    three_temp <= 'I';
    rst <= 'I';
WHEN X"4004" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';

```

```

glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '1';
two_temp <= '0';
three_temp <= '0';
rst <= '1';
WHEN X"4008" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '1';
    three_temp <= '0';
    rst <= '1';
WHEN X"400C" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '1';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '1';
    rst <= '1';

```

-- We would add the Trigger commands for channels 4 through 8 here

```

WHEN X"6000" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';

```

```

two_temp <= '0';
three_temp <= '0';
rst <= '1';
WHEN X"6004" =>
mask_zero_en <= '0';
mask_one_en <= '0';
mask_two_en <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '0';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= '1';
WHEN X"6008" =>
mask_zero_en <= '0';
mask_one_en <= '0';
mask_two_en <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '0';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '1';
three_temp <= '0';
rst <= '1';
WHEN X"600C" =>
mask_zero_en <= '0';
mask_one_en <= '0';
mask_two_en <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '0';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= '1';

```

-- We would add the Write Transducer Data command for channels 4 through 8 here

```
WHEN X"8000" =>
```

```

mask_zero_en <= '0';
mask_one_en  <= '0';
mask_two_en  <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '0';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= '0';
WHEN others =>
-- Ignore the other commands...
mask_zero_en <= '0';
mask_one_en  <= '0';
mask_two_en  <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '0';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= '1';

```

END CASE;

ELSE -- Read Operation so set the different operations...

```

internal_write <= '0';
CASE haddress(15 downto 0) IS
  WHEN X"0000" =>
    mask_zero_en <= '0';
    mask_one_en  <= '0';
    mask_two_en  <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';

```



```

        rst <= '1';
WHEN X"0004" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';
WHEN X"0008" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';
WHEN X"000C" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';

```

-- We would add the Read INT_MASK command for channels 4 through 8 here

```

WHEN X"2000" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';

```

```

mask_two_en  <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '0';
glue_zero <= '1';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= '1';
WHEN X"2004" =>
mask_zero_en <= '0';
mask_one_en  <= '0';
mask_two_en  <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '0';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= '1';
WHEN X"2008" =>
mask_zero_en <= '0';
mask_one_en  <= '0';
mask_two_en  <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '0';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= '1';
WHEN X"200C" =>
mask_zero_en <= '0';
mask_one_en  <= '0';
mask_two_en  <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '0';

```

```

glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '1';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '1';
rst <= '1';
-- We would add the Read Status command for channels 4 through 8 here
WHEN X"4000" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '1';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '1';
    one_temp <= '1';
    two_temp <= '1';
    three_temp <= '1';
    rst <= '1';
WHEN X"4004" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '1';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';
WHEN X"4008" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';

```

```

zero_temp <= '0';
one_temp <= '0';
two_temp <= '1';
three_temp <= '0';
rst <= '1';
WHEN X"400C" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '1';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '1';
    rst <= '1';
-- We would add the Trigger command for channels 4 through 8 here
WHEN X"6000" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '1';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';
WHEN X"6004" =>
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '1';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';

```

```

        rst <= '1';
WHEN X"6008" =>
    mask_zero_en <= '0';
    mask_one_en  <= '0';
    mask_two_en  <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '1';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '1';
    three_temp <= '0';
    rst <= '1';
WHEN X"600C" =>
    mask_zero_en <= '0';
    mask_one_en  <= '0';
    mask_two_en  <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';
-- We would add the Read Transducer Data command for channels 4 through 8 here
WHEN X"8000" =>
-- reset_STIM should get one
    mask_zero_en <= '0';
    mask_one_en  <= '0';
    mask_two_en  <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '1';
    rst <= '0';
WHEN others =>
    mask_zero_en <= '0';

```

```

mask_one_en  <= '0';
mask_two_en  <= '0';
mask_three_en <= '0';
zero_en <= '0';
three_en <= '0';
invalid <= '1';
glue_zero <= '0';
glue_one <= '0';
glue_two <= '0';
glue_three <= '0';
zero_temp <= '0';
one_temp <= '0';
two_temp <= '0';
three_temp <= '0';
rst <= '1';

END CASE;
END IF;
state <= data;
ELSE
    internal_write <= '0';
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';
    state <= address;
END IF;
WHEN data =>
    -- Remain in data state on burst transfers
    -- Never happens in this example but it could for bigger TIMs
    IF htrans = "11" THEN
        IF hwrite = '1' THEN
            internal_write <= '1';
        ELSE
            internal_write <= '0';
        END IF;
        mask_zero_en <= '0';
        mask_one_en <= '0';
        mask_two_en <= '0';
        mask_three_en <= '0';
        zero_en <= '0';
        three_en <= '0';
        invalid <= '0';
        glue_zero <= '0';
        glue_one <= '0';

```

```

        glue_two <= '0';
        glue_three <= '0';
        zero_temp <= '0';
        one_temp <= '0';
        two_temp <= '0';
        three_temp <= '0';
        rst <= '1';
        state <= data;
    ELSE
        internal_write <= '0';
        mask_zero_en <= '0';
        mask_one_en <= '0';
        mask_two_en <= '0';
        mask_three_en <= '0';
        zero_en <= '0';
        three_en <= '0';
        invalid <= '0';
        glue_zero <= '0';
        glue_one <= '0';
        glue_two <= '0';
        glue_three <= '0';
        zero_temp <= '0';
        one_temp <= '0';
        two_temp <= '0';
        three_temp <= '0';
        rst <= '1';
        state <= address;
    END IF;
WHEN others =>
    internal_write <= '0';
    mask_zero_en <= '0';
    mask_one_en <= '0';
    mask_two_en <= '0';
    mask_three_en <= '0';
    zero_en <= '0';
    three_en <= '0';
    invalid <= '0';
    glue_zero <= '0';
    glue_one <= '0';
    glue_two <= '0';
    glue_three <= '0';
    zero_temp <= '0';
    one_temp <= '0';
    two_temp <= '0';
    three_temp <= '0';
    rst <= '1';
    state <= address;
END CASE;
END IF;
END PROCESS;

-- Create the Register Bank
PROCESS(hclock,hresetn)
BEGIN
    IF hresetn = '0' THEN

```

```

mask_data <= (others => '1');
zero_out <= (others => '0');
three_out <= (others => '0');
haddress_reg <= (others => '0');
hrdata <= (others => '0');
rst <= '1';
ELSIF rising_edge(hclock) THEN
    haddress_reg <= haddress(31 downto 0);
    IF internal_write = '1' THEN
        CASE haddress_reg(15 downto 0) IS
            WHEN X"0000" =>
                mask_data <= hwdata;
                rst <= '1';
            WHEN X"0004" =>
                mask_data <= hwdata;
                rst <= '1';
            WHEN X"0008" =>
                mask_data <= hwdata;
                rst <= '1';
            WHEN X"000C" =>
                mask_data <= hwdata;
                rst <= '1';
            -- We would do the same for channels 4 through 8 here
            WHEN X"2000" =>
                rst <= '1';
            WHEN X"2004" =>
                rst <= '1';
            WHEN X"2008" =>
                rst <= '1';
            WHEN X"200C" =>
                rst <= '1';
            -- We would do the same for channels 4 through 8 here
            WHEN X"4000" =>
                rst <= '1';
            WHEN X"4004" =>
                rst <= '1';
            WHEN X"4008" =>
                rst <= '1';
            WHEN X"400C" =>
                rst <= '1';
            -- We would do the same for channels 4 through 8 here
            WHEN X"6000" =>
                rst <= '1';
                zero_out <= hwdata;
            WHEN X"6004" =>
                rst <= '1';
                -- no effect
            WHEN X"6008" =>
                rst <= '1';
                -- no effect
            WHEN X"600C" =>
                rst <= '1';
                three_out <= hwdata;
            -- We would do the same for channels 4 through 8 here, note that only actuator channels can be
            -- written to... So ignore if any of the other channels are sensors
            WHEN X"8000" =>

```



```

-- reset_STIM
rst <= '0';
WHEN others =>
    rst <= '1';
    null;
END CASE;

END IF;
IF hsel = '1' AND hwrite = '0' THEN
    rst <= '1';
    CASE haddress(15 downto 0) IS
        WHEN X"0000" =>
            rst <= '1';
            hrdata <= INT_MASK_ZERO;
        WHEN X"0004" =>
            rst <= '1';
            hrdata <= INT_MASK_ONE;
        WHEN X"0008" =>
            rst <= '1';
            hrdata <= INT_MASK_TWO;
        WHEN X"000C" =>
            rst <= '1';
            hrdata <= INT_MASK_THREE;
        -- We would do the same for channels 4 through 8 here
        WHEN X"2000" =>
            rst <= '1';
            hrdata <= status_zero;
        WHEN X"2004" =>
            rst <= '1';
            hrdata <= status_one;
        WHEN X"2008" =>
            rst <= '1';
            hrdata <= status_two;
        WHEN X"200C" =>
            rst <= '1';
            hrdata <= status_three;
        -- We would do the same for channels 4 through 8 here
        WHEN X"4000" =>
            rst <= '1';
            hrdata <= (others => '0');
        WHEN X"4004" =>
            rst <= '1';
            hrdata <= (others => '0');
        WHEN X"4008" =>
            rst <= '1';
            hrdata <= (others => '0');
        WHEN X"400C" =>
            rst <= '1';
            hrdata <= (others => '0');
        -- We would do the same for channels 4 through 8 here
        WHEN X"6000" =>
            rst <= '1';
            hrdata <= data_zero_in;
        WHEN X"6004" =>
            rst <= '1';
            hrdata <= data_one;
    END CASE;
END IF;

```

```

        WHEN X"6008" =>
            rst <= '1';
            hrdata <= data_two;
        WHEN X"600C" =>
            rst <= '1';
            -- do nothing
        -- We would do the same for channels 4 through 8 here, note that only sensor channels can be
        -- read from... So ignore if any of the other channels are actuators
        WHEN X"8000" =>
            -- reset_STIM
            rst <= '0';
        WHEN others =>
            rst <= '0';
            hrdata <= (others => '0');
    END CASE;
END IF;
END IF;
END PROCESS;
-- Assign all the signals
INT_MASK_ZERO_en <= mask_zero_en;
INT_MASK_DATA <= mask_data;
INT_MASK_ONE_en <= mask_one_en;
INT_MASK_TWO_en <= mask_two_en;
INT_MASK_THREE_en <= mask_three_en;
status_invalid <= invalid;
data_zero_en <= zero_en;
data_three_en <= three_en;
RD_Status_ZERO <= glue_zero;
RD_Sensor_one <= glue_one;
RD_Sensor_two <= glue_two;
RD_Status_three <= glue_three;
trigger_zero <= zero_temp;
trigger_one <= one_temp;
trigger_two <= two_temp;
trigger_three <= three_temp;
STIM_Reset <= rst;
END rtl;

```

Next, we show the VHDL code for the Transducer Block. Note that this block should not change.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY Transducer_Block IS
    PORT(
        INT_MASK_en   : IN   std_logic;
        INT_MASK_in   : IN   std_logic_vector (31 DOWNT0 0);
        Transducer_sel : IN   std_logic;
        actuator_data  : IN   std_logic_vector (31 DOWNT0 0);
        actuator_en    : IN   std_logic;
        clk            : IN   std_logic;
        reset          : IN   std_logic;
    );
END ENTITY Transducer_Block;

```

```

    sensor_data : IN  std_logic_vector (31 DOWNT0 0);
    sensor_en   : IN  std_logic;
    status_en   : IN  std_logic;
    status_in   : IN  std_logic_vector (31 DOWNT0 0);
    INT_MASK_out : OUT std_logic_vector (31 DOWNT0 0);
    INT_out      : OUT std_logic_vector (31 DOWNT0 0);
    Status_out   : OUT std_logic_vector (31 DOWNT0 0);
    data_out     : OUT std_logic_vector (31 DOWNT0 0)
);

-- Declarations

END Transducer_Block ;

ARCHITECTURE struct OF Transducer_Block IS

    -- Architecture declarations

    -- Internal signal declarations
    SIGNAL INT_MASK_reg : std_logic_vector(31 DOWNT0 0);
    SIGNAL Status_reg   : std_logic_vector(31 DOWNT0 0);
    SIGNAL data_en      : std_logic;
    SIGNAL data_in      : std_logic_vector(31 DOWNT0 0);

    -- Component Declarations
    COMPONENT reg32
    GENERIC (
        WIDTH : INTEGER
    );
    PORT (
        clk : IN  std_logic;
        din : IN  std_logic_vector (WIDTH-1 DOWNT0 0);
        en  : IN  std_logic;
        reset : IN  std_logic;
        dout : OUT std_logic_vector (WIDTH-1 DOWNT0 0)
    );
    END COMPONENT;

    -- Optional embedded configurations
    -- pragma synthesis_off
    -- FOR ALL : reg32 USE ENTITY my_project_lib.reg32;
    -- pragma synthesis_on

BEGIN

    -- Architecture concurrent statements
    -- HDL Embedded Text Block 1 eb1
    -- eb1 1
    Status_out <= status_reg;
    INT_MASK_out <= INT_MASK_reg;
    INT_out <= Status_reg AND INT_MASK_reg;

    -- HDL Embedded Text Block 3 eb3
    -- eb3 3
    Data_in <= actuator_data when Transducer_sel = '0' else

```

```

    sensor_data;

-- HDL Embedded Text Block 4 eb4
-- eb4 4
Data_en <= actuator_en when Transducer_sel = '0' else
    sensor_en;

-- Instance port mappings.
I0 : reg32
    GENERIC MAP (
        WIDTH => 32
    )
    PORT MAP (
        clk => clk,
        reset => reset,
        en => data_en,
        din => data_in,
        dout => data_out
    );
I1 : reg32
    GENERIC MAP (
        WIDTH => 32
    )
    PORT MAP (
        clk => clk,
        reset => reset,
        en => status_en,
        din => status_in,
        dout => Status_reg
    );
I2 : reg32
    GENERIC MAP (
        WIDTH => 32
    )
    PORT MAP (
        clk => clk,
        reset => reset,
        en => INT_MASK_en,
        din => INT_MASK_in,
        dout => INT_MASK_reg
    );

END struct;

```

Next, we show the VHDL code for the priority encoder.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY INT_generation IS
    PORT(
        CHANNEL_ZERO : IN    std_logic_vector (31 DOWNTO 0);

```

```

CHANNEL_1 : IN  std_logic_vector (31 DOWNTO 0);
CHANNEL_2 : IN  std_logic_vector (31 DOWNTO 0);
CHANNEL_3 : IN  std_logic_vector (31 DOWNTO 0);
CHANNEL_4 : IN  std_logic_vector (31 DOWNTO 0);
CHANNEL_5 : IN  std_logic_vector (31 DOWNTO 0);
CHANNEL_6 : IN  std_logic_vector (31 DOWNTO 0);
CHANNEL_7 : IN  std_logic_vector (31 DOWNTO 0);
CHANNEL_8 : IN  std_logic_vector (31 DOWNTO 0);
CHANNEL_9 : IN  std_logic_vector (31 DOWNTO 0);
CHANNEL_10 : IN std_logic_vector (31 DOWNTO 0);
CHANNEL_11 : IN std_logic_vector (31 DOWNTO 0);
CHANNEL_12 : IN std_logic_vector (31 DOWNTO 0);
CHANNEL_13 : IN std_logic_vector (31 DOWNTO 0);
CHANNEL_14 : IN std_logic_vector (31 DOWNTO 0);
CHANNEL_15 : IN std_logic_vector (31 DOWNTO 0);
CHANNEL_16 : IN std_logic_vector (31 DOWNTO 0);
CHANNEL_17 : IN std_logic_vector (31 DOWNTO 0);
CHANNEL_18 : IN std_logic_vector (31 DOWNTO 0);
CHANNEL_19 : IN std_logic_vector (31 DOWNTO 0);
INT       : OUT std_logic_vector (5 DOWNTO 0)
);

-- Declarations

END INT_generation ;

ARCHITECTURE struct OF INT_generation IS

BEGIN

    eb5_truth_process: PROCESS(CHANNEL_ZERO, Channel_1, Channel_2, Channel_3, Channel_4,
Channel_5, Channel_6, Channel_7, Channel_8, Channel_9, Channel_10, Channel_11, Channel_12, Channel_13,
Channel_14, Channel_15, Channel_16, Channel_17, Channel_18, Channel_19 )

BEGIN

    -- CHANNEL_ZERO definition
    IF CHANNEL_ZERO(0) = '1' THEN
        INT <= "000001";
    ELSIF CHANNEL_ZERO(1) = '1' THEN
        INT <= "000010";
    ELSIF CHANNEL_ZERO(2) = '1' THEN
        INT <= "000011";
    ELSIF CHANNEL_ZERO(3) = '1' THEN
        INT <= "000100";
    -- Channel one
    ELSIF CHANNEL_1(0) = '1' THEN
        INT <= "000101";
    ELSIF CHANNEL_1(2) = '1' THEN
        INT <= "000110";
    ELSIF CHANNEL_1(3) = '1' THEN
        INT <= "000111";

```

```

-- Channel two
ELSIF CHANNEL_2(0) = '1' THEN
    INT <= "001000";
ELSIF CHANNEL_2(2) = '1' THEN
    INT <= "001001";
ELSIF CHANNEL_2(3) = '1' THEN
    INT <= "001010";
-- Channel Three
ELSIF CHANNEL_3(0) = '1' THEN
    INT <= "001011";
ELSIF CHANNEL_3(2) = '1' THEN
    INT <= "001100";
ELSIF CHANNEL_3(3) = '1' THEN
    INT <= "001101";
-- Channel Four
ELSIF CHANNEL_4(0) = '1' THEN
    INT <= "001110";
ELSIF CHANNEL_4(2) = '1' THEN
    INT <= "001111";
ELSIF CHANNEL_4(3) = '1' THEN
    INT <= "010000";
-- Channel Five
ELSIF CHANNEL_5(0) = '1' THEN
    INT <= "010001";
ELSIF CHANNEL_5(2) = '1' THEN
    INT <= "010010";
ELSIF CHANNEL_5(3) = '1' THEN
    INT <= "010011";
-- Channel Six
ELSIF CHANNEL_6(0) = '1' THEN
    INT <= "010100";
ELSIF CHANNEL_6(2) = '1' THEN
    INT <= "010101";
ELSIF CHANNEL_6(3) = '1' THEN
    INT <= "010110";
-- Channel Seven
ELSIF CHANNEL_7(0) = '1' THEN
    INT <= "010111";
ELSIF CHANNEL_7(2) = '1' THEN
    INT <= "011000";
ELSIF CHANNEL_7(3) = '1' THEN
    INT <= "011001";
-- Channel Eight
ELSIF CHANNEL_8(0) = '1' THEN
    INT <= "011010";
ELSIF CHANNEL_8(2) = '1' THEN
    INT <= "011011";
ELSIF CHANNEL_8(3) = '1' THEN
    INT <= "011100";
-- Channel Nine
ELSIF CHANNEL_9(0) = '1' THEN
    INT <= "011101";
ELSIF CHANNEL_9(2) = '1' THEN
    INT <= "011110";
ELSIF CHANNEL_9(3) = '1' THEN
    INT <= "011111";

```

```

-- Channel Ten
ELSIF CHANNEL_10(0) = '1' THEN
    INT <= "100000";
ELSIF CHANNEL_10(2) = '1' THEN
    INT <= "100001";
ELSIF CHANNEL_10(3) = '1' THEN
    INT <= "100010";
-- Channel Eleven
ELSIF CHANNEL_11(0) = '1' THEN
    INT <= "100011";
ELSIF CHANNEL_11(2) = '1' THEN
    INT <= "100100";
ELSIF CHANNEL_11(3) = '1' THEN
    INT <= "100101";
-- Channel Twelve
ELSIF CHANNEL_12(0) = '1' THEN
    INT <= "100110";
ELSIF CHANNEL_12(2) = '1' THEN
    INT <= "100111";
ELSIF CHANNEL_12(3) = '1' THEN
    INT <= "101000";
-- Channel Thirteen
ELSIF CHANNEL_13(0) = '1' THEN
    INT <= "101001";
ELSIF CHANNEL_13(2) = '1' THEN
    INT <= "101010";
ELSIF CHANNEL_13(3) = '1' THEN
    INT <= "101011";
-- Channel Fourteen
ELSIF CHANNEL_14(0) = '1' THEN
    INT <= "101100";
ELSIF CHANNEL_14(2) = '1' THEN
    INT <= "101101";
ELSIF CHANNEL_14(3) = '1' THEN
    INT <= "101110";
-- Channel Fifteen
ELSIF CHANNEL_15(0) = '1' THEN
    INT <= "101111";
ELSIF CHANNEL_15(2) = '1' THEN
    INT <= "110000";
ELSIF CHANNEL_15(3) = '1' THEN
    INT <= "110001";
-- Channel Sixteen
ELSIF CHANNEL_16(0) = '1' THEN
    INT <= "110010";
ELSIF CHANNEL_16(2) = '1' THEN
    INT <= "110011";
ELSIF CHANNEL_16(3) = '1' THEN
    INT <= "110100";
-- Channel Seventeen
ELSIF CHANNEL_17(0) = '1' THEN
    INT <= "110101";
ELSIF CHANNEL_17(2) = '1' THEN
    INT <= "110110";
ELSIF CHANNEL_17(3) = '1' THEN
    INT <= "110111";

```

```

-- Channel Eighteen
ELSIF CHANNEL_18(0) = '1' THEN
    INT <= "111000";
ELSIF CHANNEL_18(2) = '1' THEN
    INT <= "111001";
ELSIF CHANNEL_18(3) = '1' THEN
    INT <= "111010";
-- Channel Nineteen
ELSIF CHANNEL_19(0) = '1' THEN
    INT <= "111011";
ELSIF CHANNEL_19(2) = '1' THEN
    INT <= "111100";
ELSIF CHANNEL_19(3) = '1' THEN
    INT <= "111101";
-- Here we would add more channels...
ELSE
    INT <= "000000";
END IF;

END PROCESS eb5_truth_process;

END struct;

```


APPENDIX D. GLUE LOGIC VHDL CODE

This Appendix contains the VHDL code for the generic “glue” logic blocks that were shown in the design chapter. First we show the VHDL for the “glue” logic for a sensor channel that is interfaced with an 8-bit ADC.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY glue_sensor_adc IS
  PORT(
    RD_sensor    : IN  std_logic;
    clk          : IN  std_logic;
    conversion_done : IN  std_logic;
    rst          : IN  std_logic;
    -- Width Size would change if ADC was bigger than 8 bits
    sensor_in    : IN  std_logic_vector (7 DOWNTO 0);
    trigger      : IN  std_logic;
    sensor_en     : OUT std_logic;
    sensor_out    : OUT std_logic_vector (31 DOWNTO 0);
    status_en     : OUT std_logic;
    status_out    : OUT std_logic_vector (31 DOWNTO 0)
  );

  -- Declarations

END glue_sensor_adc ;

--
-- VHDL Architecture my_project_lib.glue_sensor_adc.struct
--
-- Created:
--   by - gustavo.UNKNOWN (ICET)
--   at - 23:05:50 06/21/2004
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE struct OF glue_sensor_adc IS

  -- Architecture declarations
```



```

        csm1_current_state <= csm1_next_state;
        -- Default Assignment To Internals

    END IF;

END PROCESS csm1_clocked;

```

```

-----
csm1_nextstate : PROCESS (
    conversion_done,
    csm1_current_state,
    trigger
)

```

```

-----
BEGIN
    CASE csm1_current_state IS
    WHEN s0 =>
        IF (trigger = '1') THEN
            csm1_next_state <= s1;
        ELSE
            csm1_next_state <= s0;
        END IF;
    WHEN s1 =>
        IF (conversion_done = '1') THEN
            csm1_next_state <= s0;
        ELSE
            csm1_next_state <= s1;
        END IF;
    WHEN OTHERS =>
        csm1_next_state <= s0;
    END CASE;

```

```

END PROCESS csm1_nextstate;

```

```

-----
csm1_output : PROCESS (
    conversion_done,
    csm1_current_state,
    trigger
)

```

```

-----
BEGIN
    -- Default Assignment
    sensor_en <= '0';
    -- Default Assignment To Internals

    -- Combined Actions
    CASE csm1_current_state IS
    WHEN s0 =>
        IF (trigger = '1') THEN
            sensor_en <= '0';
        ELSE
            sensor_en <= '0';
        END IF;
    WHEN s1 =>
        IF (conversion_done = '1') THEN

```

```

        sensor_en <= '1';
    ELSE
        sensor_en <= '0';
    END IF;
    WHEN OTHERS =>
        NULL;
    END CASE;

END PROCESS csm1_output;

-- Concurrent Statements

-- HDL Embedded Text Block 3 eb1
-- eb1 3
-- If ADC is 10 or 12 bits, then we would have to change this block
sensor_out <= X"000000" & sensor_in;

-- Instance port mappings.

END struct;

```

Next, is the “glue” logic for sensors that are interfaced with Digital IO.

```

ENTITY glue_sensor_digitalio IS
    PORT(
        RD_sensor : IN    std_logic;
        clk       : IN    std_logic;
        rst       : IN    std_logic;
        -- Width size varies depending on the amount of digital I/O that the sensor is interfaced with
        sensor_in  : IN    std_logic_vector (7 DOWNTO 0);
        trigger    : IN    std_logic;
        sensor_en  : OUT    std_logic;
        sensor_out : OUT    std_logic_vector (31 DOWNTO 0);
        status_en  : OUT    std_logic;
        status_out : OUT    std_logic_vector (31 DOWNTO 0)
    );

-- Declarations

END glue_sensor_digitalio ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE struct OF glue_sensor_digitalio IS

    -- Architecture declarations
    -- Non hierarchical state machine declarations
    TYPE CSM2_STATE_TYPE IS (
        s2,

```



```

        csm2_current_state <= s2;
        -- Reset Values
    ELSIF (clk'EVENT AND clk = '1') THEN
        csm2_current_state <= csm2_next_state;
        -- Default Assignment To Internals

    END IF;

END PROCESS csm2_clocked;

-----

csm2_nextstate : PROCESS (
    csm2_current_state,
    trigger
)
-----

BEGIN
    CASE csm2_current_state IS
    WHEN s2 =>
        IF (trigger = '1') THEN
            csm2_next_state <= s3;
        ELSE
            csm2_next_state <= s2;
        END IF;
    WHEN s3 =>
        csm2_next_state <= s2;
    WHEN OTHERS =>
        csm2_next_state <= s2;
    END CASE;

END PROCESS csm2_nextstate;

-----

csm2_output : PROCESS (
    csm2_current_state,
    trigger
)
-----

BEGIN
    -- Default Assignment
    sensor_en <= '0';
    sensor_out <= "00000000000000000000000000000000";
    -- Default Assignment To Internals

    -- Combined Actions
    CASE csm2_current_state IS
    WHEN s2 =>
        IF (trigger = '1') THEN
            sensor_en <= '0';
            done <= '0';
        ELSE
            sensor_en <= '0';
            done <= '0';
        END IF;
    WHEN s3 =>
        sensor_en <= '1';

```

```

        done <= '1';
    WHEN OTHERS =>
        NULL;
    END CASE;

END PROCESS csm2_output;

-- Concurrent Statements

-- Instance port mappings.

END struct;

```

Next, we show the VHDL code for an actuator channel that is interface with an 8-bit DAC.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY glue_actuator_dac IS
    PORT(
        RD_status      : IN  std_logic;
        actuator_in     : IN  std_logic_vector (31 DOWNTO 0);
        clk             : IN  std_logic;
        conversion_done : IN  std_logic;
        rst             : IN  std_logic;
        trigger         : IN  std_logic;
        status_en       : OUT  std_logic;
        status_out      : OUT  std_logic_vector (31 DOWNTO 0);
        -- Width size may vary depending on the size of the DAC
        to_actuator     : OUT  std_logic_vector (7 DOWNTO 0)
    );

-- Declarations

END glue_actuator_dac ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

LIBRARY my_project_lib;

ARCHITECTURE struct OF glue_actuator_dac IS

    -- Architecture declarations
    -- Non hierarchical state machine declarations
    TYPE CSM2_STATE_TYPE IS (

```

```

    s2,
    s3
);

-- State vector declaration
ATTRIBUTE state_vector : string;
ATTRIBUTE state_vector OF struct : ARCHITECTURE IS "csm2_current_state" ;

-- Declare current and next state signals
SIGNAL csm2_current_state : CSM2_STATE_TYPE ;
SIGNAL csm2_next_state : CSM2_STATE_TYPE ;

-- Internal signal declarations
SIGNAL actuator_en : std_logic;
SIGNAL actuator_out : std_logic_vector(7 DOWNTO 0);

-- Component Declarations
COMPONENT reg32
GENERIC (
    WIDTH : INTEGER
);
PORT (
    clk : IN    std_logic;
    din  : IN    std_logic_vector (WIDTH-1 DOWNTO 0);
    en   : IN    std_logic;
    reset : IN    std_logic;
    dout : OUT    std_logic_vector (WIDTH-1 DOWNTO 0)
);
END COMPONENT;

-- Optional embedded configurations
-- pragma synthesis_off
FOR ALL : reg32 USE ENTITY my_project_lib.reg32;
-- pragma synthesis_on

BEGIN
    -- Architecture concurrent statements
    -- HDL Embedded Text Block 1 eb1
    -- eb1 1
    -- May change depending on the width size of the DAC
    actuator_out <= actuator_in (7 downto 0);

    -- HDL Embedded Text Block 2 status_generation1
    process (clk, rst, RD_status)
    begin
        if rst = '0' then
            status_en <= '0';
            status_out <= (others => '0');
        elsif (rising_edge(clk)) then
            status_en <= '1';
            -- Trigger ACK
            if conversion_done = '1' then

```



```
END PROCESS csm2_nextstate;
```

```
-----  
csm2_output : PROCESS (  
    conversion_done,  
    csm2_current_state,  
    trigger  
)  
-----
```

```
BEGIN
```

```
-- Default Assignment  
actuator_en <= '0';  
-- Default Assignment To Internals
```

```
-- Combined Actions
```

```
CASE csm2_current_state IS
```

```
WHEN s2 =>
```

```
    IF (trigger = '1') THEN
```

```
        actuator_en <= '1';
```

```
    ELSE
```

```
        actuator_en <= '0';
```

```
    END IF;
```

```
WHEN s3 =>
```

```
    IF (conversion_done = '1') THEN
```

```
        actuator_en <= '0';
```

```
    ELSE
```

```
        actuator_en <= '0';
```

```
    END IF;
```

```
WHEN OTHERS =>
```

```
    NULL;
```

```
END CASE;
```

```
END PROCESS csm2_output;
```

```
-- Concurrent Statements
```

```
-- Instance port mappings.
```

```
I0 : reg32
```

```
    GENERIC MAP (  
        WIDTH => 8
```

```
    )
```

```
    PORT MAP (  
        clk => clk,
```

```
        reset => rst,
```

```
        en => actuator_en,
```

```
        din => actuator_out,
```

```
        dout => to_actuator
```

```
    );
```

```
END struct;
```

Next, we show the VHDL code for an actuator channel that is interfaced with digital I/O.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY glue_actuator_digitalio IS
  PORT(
    RD_status : IN  std_logic;
    actuator_in : IN  std_logic_vector (31 DOWNT0 0);
    clk       : IN  std_logic;
    rst       : IN  std_logic;
    trigger    : IN  std_logic;
    status_en  : OUT std_logic;
    status_out : OUT std_logic_vector (31 DOWNT0 0);
    -- Width size varies depending on the amount of digital I/O that is interfaced with the actuator
    to_actuator : OUT std_logic
  );

  -- Declarations

END glue_actuator_digitalio ;

--
-- VHDL Architecture my_project_lib.glue_actuator_digitalio.struct
--
-- Created:
--   by - gustavo.UNKNOWN (ICET)
--   at - 23:22:52 06/21/2004
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2003.1 (Build 399)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE struct OF glue_actuator_digitalio IS

  -- Architecture declarations
  -- Non hierarchical state machine declarations
  TYPE CSM3_STATE_TYPE IS (
    s4,
    s5
  );

  -- State vector declaration
  ATTRIBUTE state_vector : string;
  ATTRIBUTE state_vector OF struct : ARCHITECTURE IS "csm3_current_state" ;

  -- Declare current and next state signals
  SIGNAL csm3_current_state : CSM3_STATE_TYPE ;
  SIGNAL csm3_next_state : CSM3_STATE_TYPE ;

  -- Internal signal declarations
  SIGNAL actuator_en : std_logic;

```

```

BEGIN
-- Architecture concurrent statements
-- HDL Embedded Text Block 1 eb1
-- eb1 1
-- This changes depending on the amount of digital I/O that is interfaced with the actuator
actuator_out <= actuator_in(0);

-- HDL Embedded Text Block 2 eb2
-- eb2 2

process(clk, rst, actuator_out)

begin
    if rst = '0' then -- Active low Asynchronous reset
        to_actuator <= '0';
    elsif (clk'event and clk = '1') then
        if actuator_en = '1' then
            to_actuator <= actuator_out;
        end if;
    end if;
end process;

-- HDL Embedded Text Block 3 status_generation2
process (clk, rst, RD_status)
begin
    if rst = '0' then
        status_en <= '0';
        status_out <= (others => '0');
    elsif (rising_edge(clk)) then
        status_en <= '1';
        -- Trigger ACK
        if done = '1' then
            status_out <= "00000000000000000000000000000101";
            -- Clear trigger ACK
        elsif RD_status = '1' then
            status_out <= "00000000000000000000000000000100";
        else
            status_out <= "00000000000000000000000000000100";
        end if;
    end if;
end process;

-- HDL Embedded Block 4 eb4
-- Non hierarchical state machine
-----
csm3_clocked : PROCESS(
    clk,
    rst
)

```

```

BEGIN
  IF (rst = '0') THEN
    csm3_current_state <= s4;
    -- Reset Values
  ELSIF (clk'EVENT AND clk = '1') THEN
    csm3_current_state <= csm3_next_state;
    -- Default Assignment To Internals

  END IF;

END PROCESS csm3_clocked;

```

```

-----
csm3_nextstate : PROCESS (
  csm3_current_state,
  trigger
)
-----

```

```

BEGIN
  CASE csm3_current_state IS
    WHEN s4 =>
      IF (trigger = '1') THEN
        csm3_next_state <= s5;
      ELSE
        csm3_next_state <= s4;
      END IF;
    WHEN s5 =>
      csm3_next_state <= s4;
    WHEN OTHERS =>
      csm3_next_state <= s4;
    END CASE;

```

```

END PROCESS csm3_nextstate;

```

```

-----
csm3_output : PROCESS (
  csm3_current_state,
  trigger
)
-----

```

```

BEGIN
  -- Default Assignment
  actuator_en <= '0';
  done <= '0';
  -- Default Assignment To Internals

  -- Combined Actions
  CASE csm3_current_state IS
    WHEN s4 =>
      IF (trigger = '1') THEN
        actuator_en <= '1';
        done <= '0';
      ELSE
        actuator_en <= '0';
        done <= '0';
      END IF;

```

```

    WHEN s5 =>
        actuator_en <= '0';
        done <= '1';
    WHEN OTHERS =>
        NULL;
    END CASE;

END PROCESS csm3_output;

-- Concurrent Statements

```

Next, we show the “glue” logic for CHANNEL_ZERO. Note that this CHANNEL_ZERO configuration is for the 3-Channel TIM that was built as the proof of concept application. However, we have commented out the different lines that can be used to expand this “glue” logic for an 8-Channel TIM implementation.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY glue_CHANNEL_ZERO IS
    PORT(
        RD_status      : IN  std_logic;
        actuator_in     : IN  std_logic_vector (31 DOWNTO 0);
        clk             : IN  std_logic;
        conversion_done  : IN  std_logic;
        conversion_done1 : IN  std_logic;
        conversion_done2 : IN  std_logic;
        -- conversion_done3 : IN  std_logic;
        -- conversion_done4 : IN  std_logic;
        -- conversion_done5 : IN  std_logic;
        -- conversion_done6 : IN  std_logic;
        -- conversion_done7 : IN  std_logic;
        invalid         : IN  std_logic;
        rst             : IN  std_logic;
        sensor_in       : IN  std_logic_vector (7 DOWNTO 0);
        -- sensor_in1      : IN  std_logic_vector (7 DOWNTO 0);
        sensor_in2      : IN  std_logic;
        --sensor_in3       : IN  std_logic;
        trigger         : IN  std_logic;
        sensor_en       : OUT  std_logic;
        sensor_out      : OUT  std_logic_vector (31 DOWNTO 0);
        status_en       : OUT  std_logic;
        status_out      : OUT  std_logic_vector (31 DOWNTO 0);
        to_actuator1    : OUT  std_logic_vector (2 DOWNTO 0);
        -- to_actuator2    : OUT  std_logic_vector (7 DOWNTO 0);
        -- to_actuator3    : OUT  std_logic;
        -- to_actuator4    : OUT  std_logic
    );

```

```

-- Declarations

END glue_CHANNEL_ZERO ;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE struct OF glue_CHANNEL_ZERO IS

    -- Architecture declarations
    -- Non hierarchical state machine declarations
    TYPE CSM2_STATE_TYPE IS (
        s2,
        s3,
        s0
    );

    -- State vector declaration
    ATTRIBUTE state_vector : string;
    ATTRIBUTE state_vector OF struct : ARCHITECTURE IS "csm2_current_state" ;

    -- Declare current and next state signals
    SIGNAL csm2_current_state : CSM2_STATE_TYPE ;
    SIGNAL csm2_next_state : CSM2_STATE_TYPE ;

    -- Internal signal declarations
    SIGNAL done : std_logic;

BEGIN
    -- Architecture concurrent statements
    -- HDL Embedded Text Block 2 status_generation1
    process (clk, rst, RD_status, invalid)
    begin
        if rst = '0' then
            status_en <= '0';
            status_out <= "00000000000000000000000000000000100";
        elsif (rising_edge(clk)) then
            status_en <= '1';
            -- Trigger ACK
            if done = '1' then
                status_out <= "00000000000000000000000000000000101";
            -- Clear trigger ACK
            elsif RD_status = '1' then
                status_out <= "00000000000000000000000000000000100";
            -- invalid command
            elsif invalid = '1' then
                status_out <= "00000000000000000000000000000000110";
            end if;
        end if;
    end process;

```

```

end process;

-- HDL Embedded Block 3 control1
-- Non hierarchical state machine
-----
csm2_clocked : PROCESS(
    clk,
    rst
)
-----
BEGIN
    IF (rst = '0') THEN
        csm2_current_state <= s2;
        -- Reset Values
    ELSIF (clk'EVENT AND clk = '1') THEN
        csm2_current_state <= csm2_next_state;
        -- Default Assignment To Internals

    END IF;

END PROCESS csm2_clocked;

-----
csm2_nextstate : PROCESS (
    conversion_done,
    conversion_done1,
    conversion_done2,
--    conversion_done3,
--    conversion_done4,
--    conversion_done5,
--    conversion_done6,
--    conversion_done7,

    csm2_current_state,
    trigger
)
-----
BEGIN
    CASE csm2_current_state IS
    WHEN s2 =>
        IF (trigger = '1') THEN
            csm2_next_state <= s3;
        ELSE
            csm2_next_state <= s2;
        END IF;
    WHEN s3 =>
        IF (conversion_done = '1' AND
            conversion_done1 = '1' AND
            conversion_done2 = '1') THEN
-- AND
--    conversion_done3 = '1' AND
--    conversion_done4 = '1' AND
--    conversion_done5 = '1' AND
--    conversion_done6 = '1' AND
--    conversion_done7 = '1') THEN
            csm2_next_state <= s0;

```



```

        ELSE
            csm2_next_state <= s3;
        END IF;
    WHEN s0 =>
        csm2_next_state <= s2;
    WHEN OTHERS =>
        csm2_next_state <= s2;
    END CASE;

END PROCESS csm2_nextstate;

-----

csm2_output : PROCESS (
    conversion_done,
    conversion_done1,
    conversion_done2,
--    conversion_done3,
--    conversion_done4,
--    conversion_done5,
--    conversion_done6,
--    conversion_done7,
    csm2_current_state,
    trigger
)
-----

BEGIN
    -- Default Assignment
    sensor_en <= '0';
    -- Default Assignment To Internals

    -- Combined Actions
    CASE csm2_current_state IS
    WHEN s2 =>
        IF (trigger = '1') THEN
            sensor_en <= '0';
            done <= '0';
        ELSE
            sensor_en <= '0';
            done <= '0';
        END IF;
    WHEN s3 =>
        IF (conversion_done = '1' AND
            conversion_done1 = '1' AND
            conversion_done2 = '1') THEN
--            conversion_done3 = '1' AND
--            conversion_done4 = '1' AND
--            conversion_done5 = '1' AND
--            conversion_done6 = '1' AND
--            conversion_done7 = '1') THEN
            sensor_en <= '1';
            done <= '1';
        ELSE
            sensor_en <= '0';
            done <= '0';
        END IF;
    WHEN s0 =>

```

```

        sensor_en <= '0';
        done <= '0';
    WHEN OTHERS =>
        NULL;
    END CASE;

END PROCESS csm2_output;

-- Concurrent Statements

-- HDL Embedded Text Block 4 eb2
-- eb1 3
sensor_out <= "000000000000000000000000" & sensor_in2 & sensor_in;
to_actuator1 <= actuator_in (2 downto 0);
-- to_actuator2 <= actuator_in (15 downto 8);
-- to_actuator3 <= actuator_in(16);
-- to_actuator4 <= actuator_in(17);

-- Instance port mappings.

END struct;

```

APPENDIX E. TEDS BLOCKS

This section shows the tables with the generated information for the TEDS blocks for the TIM that was implemented. First, we show the Meta-TEDS implemented block with all its values.

Field No.	Description	Value
1	Meta-TEDS Length	20
2	IEEE 1451 Standards Family Working Group Number	255
3	TEDS Version Number	255
4	Number of Implemented Channels	3
5	Worst-Case Channel Data Model Length	8
6	Worst-Case Channel Update Time (t_{wu})	2.7×10^{-6}
7	Worst-Case Channel Sampling Period (t_{wsp})	2.5×10^{-6}
8	Channel Groupings Data Sub-block Length	0
9	Number of Channel Groupings = G	-
10	Group Type	-
11	Number of Group Members = N	-
12	Member Channel Numbers List = M(N)	-
13	Checksum for Meta-TEDS	64210

The following table shows the implementation of the temperature sensor transducer channel.

Field No.	Description	Value
1	Channel TEDS Length	42
2	Calibration Key	4
3	Channel Type Key	0
4	Physical Units	Celsius (0, 128, 128, 128, 128, 128, 128, 128, 130, 128, 128)
5	Lower Range Limit	16
6	Upper Range Limit	48
7	Worst-Case Uncertainty	0.5
8	Channel Data Model	0

9	Channel Data Model Length	1
10	Channel Model Significant Bits	8
11	Channel Update Time (t_u)	2.7×10^{-6}
12	Channel Sampling Period (t_{sp})	2.5×10^{-6}
13	Checksum for Channel TEDS	63352

The next table shows the Channel-TEDS for the light sensor.

Field No.	Description	Value
1	Channel TEDS Length	42
2	Calibration Key	0
3	Channel Type Key	0
4	Physical Units	Boolean value (4, 128, 128, 128, 128, 128, 128, 128, 128)
5	Lower Range Limit	0
6	Upper Range Limit	1
7	Worst-Case Uncertainty	0
8	Channel Data Model	0
9	Channel Data Model Length	1
10	Channel Model Significant Bits	1
11	Channel Update Time (t_u)	2.5×10^{-8}
12	Channel Sampling Period (t_{sp})	2.5×10^{-8}
13	Checksum for Channel TEDS	63126

Lastly, the following table shows the Channel-TEDS for the LEDs.

Field No.	Description	Value
1	Channel TEDS Length	42
2	Calibration Key	0
3	Channel Type Key	1
4	Physical Units	Boolean Bit Sequence (4, 128, 128, 128, 128, 128, 128, 128, 128)
5	Lower Range Limit	0
6	Upper Range Limit	15
7	Worst-Case Uncertainty	0
8	Channel Data Model	0
9	Channel Data Model Length	1
10	Channel Model Significant Bits	4
11	Channel Update Time (t_u)	2.5×10^{-8}

12	Channel Sampling Period (t_{sp})	2.5×10^{-8}
13	Checksum for Channel TEDS	63108

BIBLIOGRAPHY

1. <http://grouper.ieee.org/groups/1451/0/>
2. IEEE Standard for a Smart Transducer Interface for Sensors and Actuators-Network Capable Application Processor (NCAP) Information Model, IEEE1451.1-1999, June 26, 1999
3. IEEE Standard for a Smart Transducer Interface for Sensors and Actuators-Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats, IEEE std 1451.2-1997, September 26, 1997
4. IEEE Standard for a Smart Transducer Interface for Sensors and Actuators-Digital Communication and Transducer Electronic Data Sheet (TEDS) Formats for Distributed Multidrop Systems. IEEE std 1451.3-2003, March 31, 2004
5. <http://grouper.ieee.org/groups/1451/4/>
6. <http://grouper.ieee.org/groups/1451/5/>
7. <http://standards.ieee.org/board/nes/projects/1451-6.pdf>
8. <http://www.analog.com/>
9. MicroConverter, Multichannel 12-bit ADC with Embedded Flash MCU. Analog Devices, 2003
10. Cummins T. et al, "An IEEE 1451 Standard Interface Chip with 12-b ADC, two 12-b DAC's, 10-kB Flash EEPROM, and 8-b Microcontroller", IEEE Journal of Solid-State Circuits, Vol. 33, No. 12, pp. 2112-2120, Dec. 1998
11. Paul Conway, Donal Heffernan, Brian O'Mara, Prof. Phil Burton, and Tremont Miao, "IEEE 1451.2: An Interpretation and Example Implementation", Instrumentation and Measurement Technology Proceedings of the 17th IEEE, vol. 2, pp. 535-540, May 2000
12. Deepika Devarajan, and Steven B. Bibyk, "A VHDL Software Model for Networking Smart Transducers through Bluetooth Technology", IEEE 2001
13. C. Girerd, S. Gardien, J. Burch, S. Katsanevas, J.Marteau, "Ethernet Network-based DAQ and Smart Sensors for the OPERA Long-baseline Neutrino Experiment", Nuclear Science Symposium Conference Record, vol.2, pp. 12/111 – 12/115, October 2000

14. L. Camara, O. Ruiz, J. Samitier, "Complete IEEE-1451 Node, STIM and NCAP, implemented for a CAN Network", IEEE Instrumentation and Measurement Technology Conference, vol.2, pp. 541-545, IMTC 2000
15. D. Wobschall, "An Implementation of IEEE 1451 NCAP for Internet Access of Serial Port-Based Sensors", Sensors for Industry Confence, pp. 157-160, Nov 2002
16. A. de Castro, T. Riesgo, E. de la Torre, Y. Torroja, J. Uceda, "Custom Hardware IEEE 1451.2 Implementation for Smart Transducers"
17. A. de Castro, J.M. Chaquet, E. Morejon, T. Riesgo, J. Uceda, "A System-on-Chip for Smart Sensors", Proceedings of the 2002 IEEE International Symposium on Industrial Electronics, vol. 2, pp. 595-599, ISIE 2002
18. Excalibur Devices Hardware Reference Manual, Version 3.1, Altera Corporation, November 2002
19. AMBA Specification, revision 2.0, ARM, 1999
20. <http://www.arm.com/>
21. Kang Lee, Eugene Song, "UML model for the IEEE 1451.1 standard", Proceedings of the 20th IEEE Instrumentation and Measurement Technology Conference, vol. 2, pp. 1587-1592, IMTC 2003
22. NIST Technical Note 1297, 1994 Edition: "Guidelines foe Evaluating and Expressing the Uncertainty of NIST Measurement Results" by Barry N. Taylor and Chris E. Kuyatt.
23. http://www.altera.com/products/ip/iup/ethernet/m-cas-mac_ethernet.html
24. http://www.altera.com/products/ip/iup/can/m-bos-c_can.html
25. <http://www.altera.com/products/ip/iup/usb/m-cas-cusb.html>
26. Standard Micro Systems Corporation, 10/100 Non-PCI Ethernet Single Chip MAC + PHY Data Sheet, Part No. LAN91C111, December 2003
27. Jameco Electronics, NTC-102 Thermistor Data Sheet
28. <http://www.national.com>
29. National Semiconductor, ADC0820 "8-Bit High Speed μ P Compatible A/D Converter with Track/Hold Function" Data Sheet, June 1999
30. Jameco Electronics, Photo Conductive Cell Part No. CDS-001 Data Sheet

31. Gene F. Franklin, J. David Powell, Michael Workman, "Digital Control of Dynamic Systems", Third Edition, Addison Wesley Longman, Inc., 1998
32. Douglas J. Smith, "HDL Chip Design", Ninth Printing, Doone Publications, July 2001
33. Brian W. Kernighan, Dennis M. Ritchie, "The C Programming Language", Second Edition, Prentice Hall, 1998
34. Randy Frank, "Understanding Smart Sensors", Second Ed., Artech House, April 2000
35. Wayne Wolf, "Computers as Components", Morgan Kaufmann Publishers, 2001